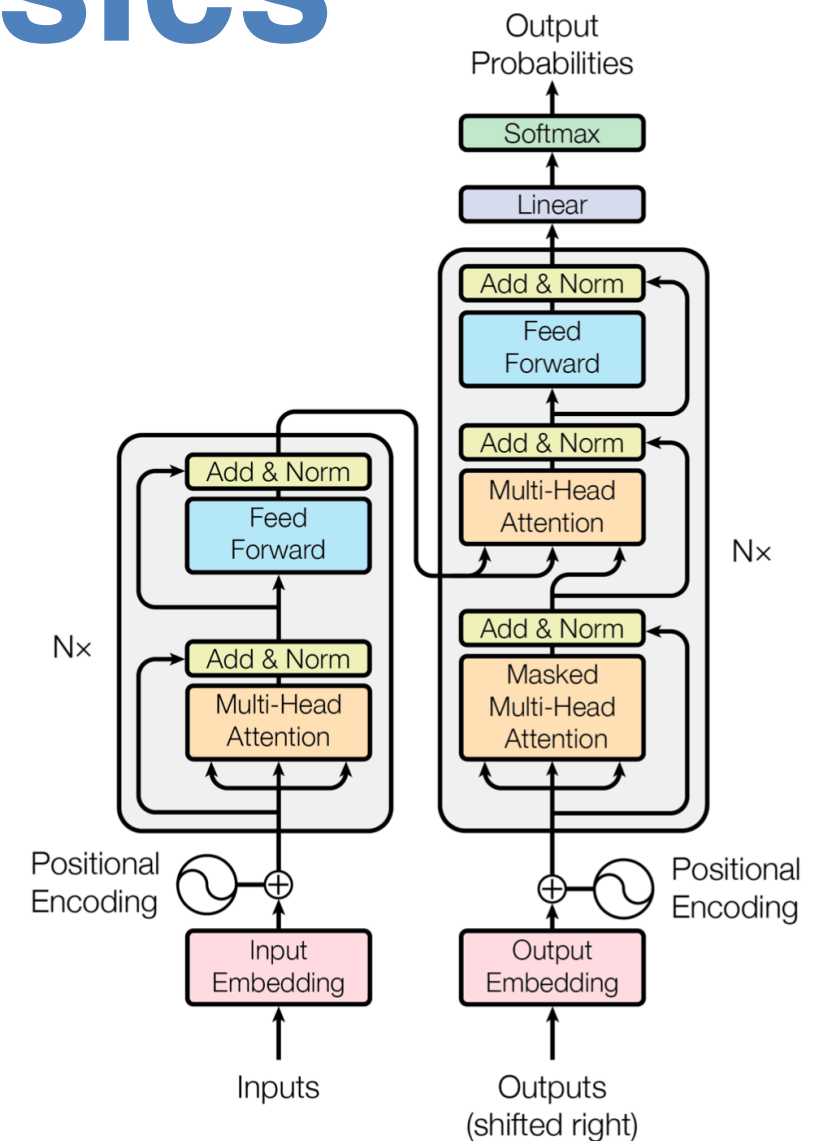


The Transformer and its uses in High Energy Physics



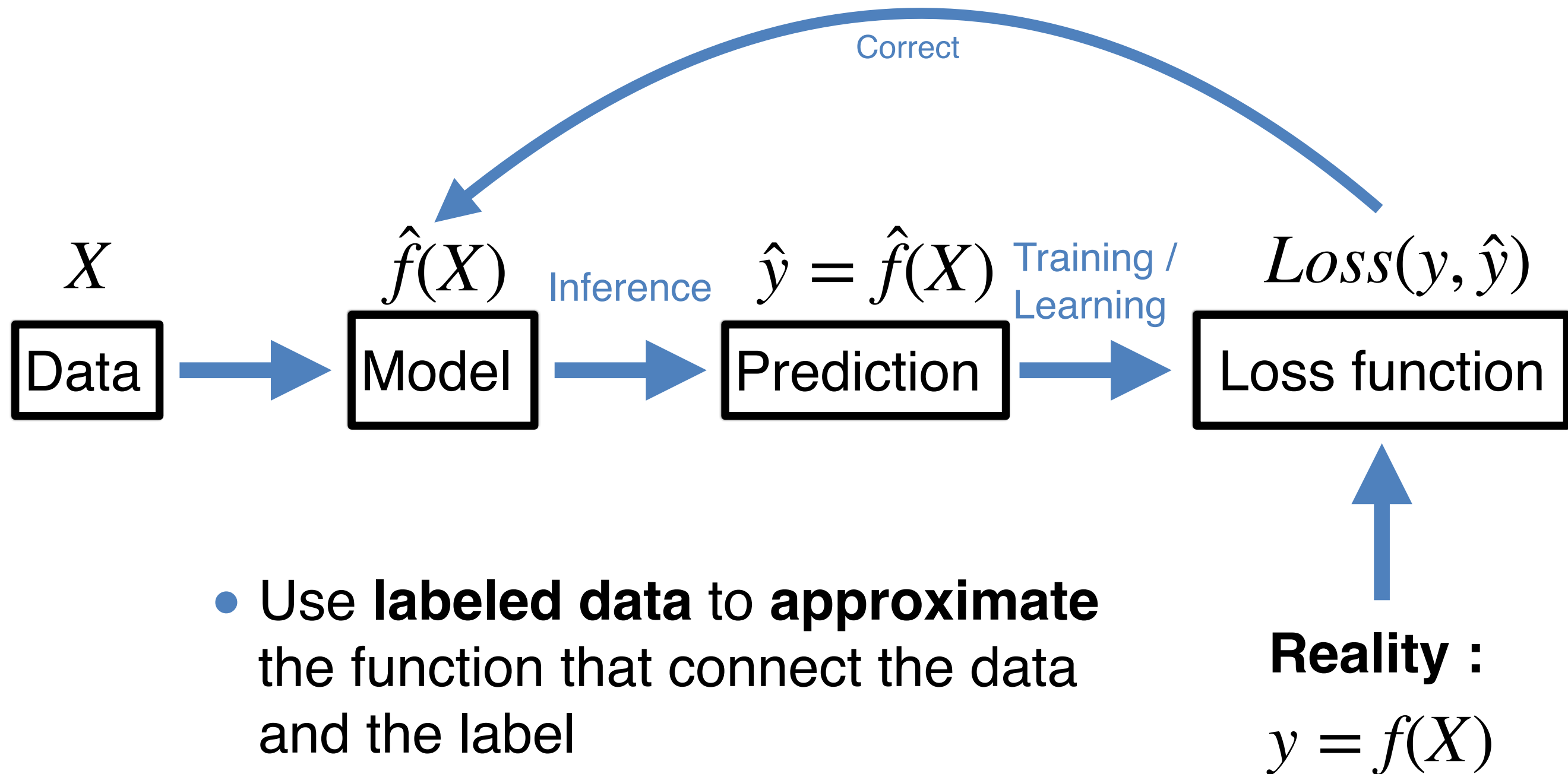
 Corentin Allaire



Timetable

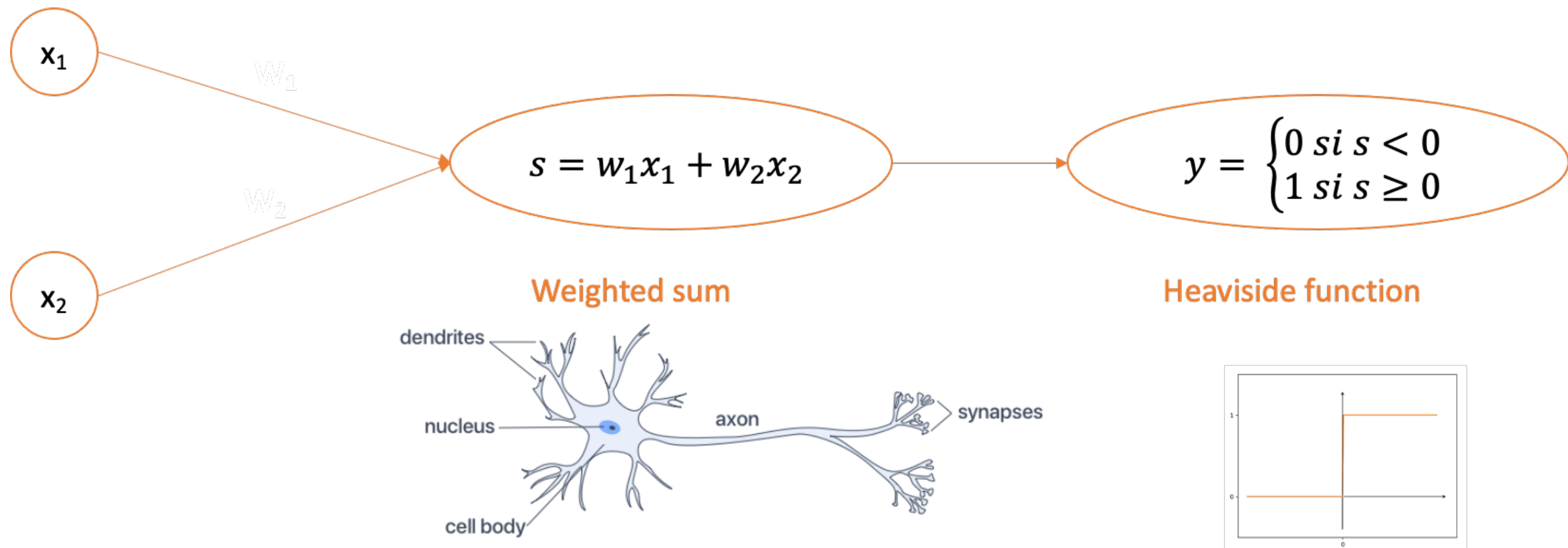
- A quick reminder on Deep Learning
- The original Transformer architecture
- Some consideration on the use of Transformers
- Examples of use in HEP

Machine Learning / Deep Learning



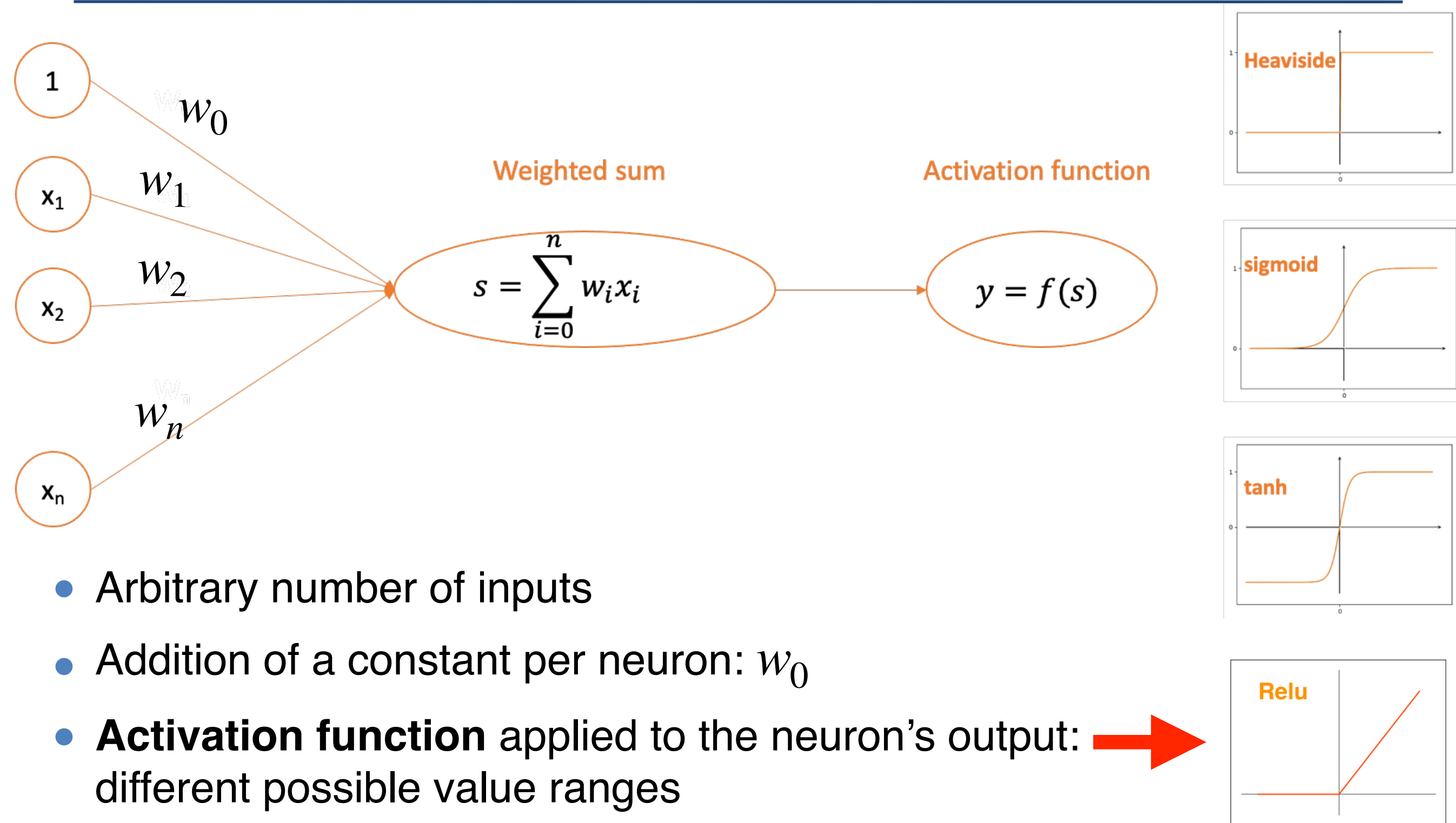
Deep Learning: The formal neuron

Idea: Create a mathematical construct that "**simulates**" biological neurons

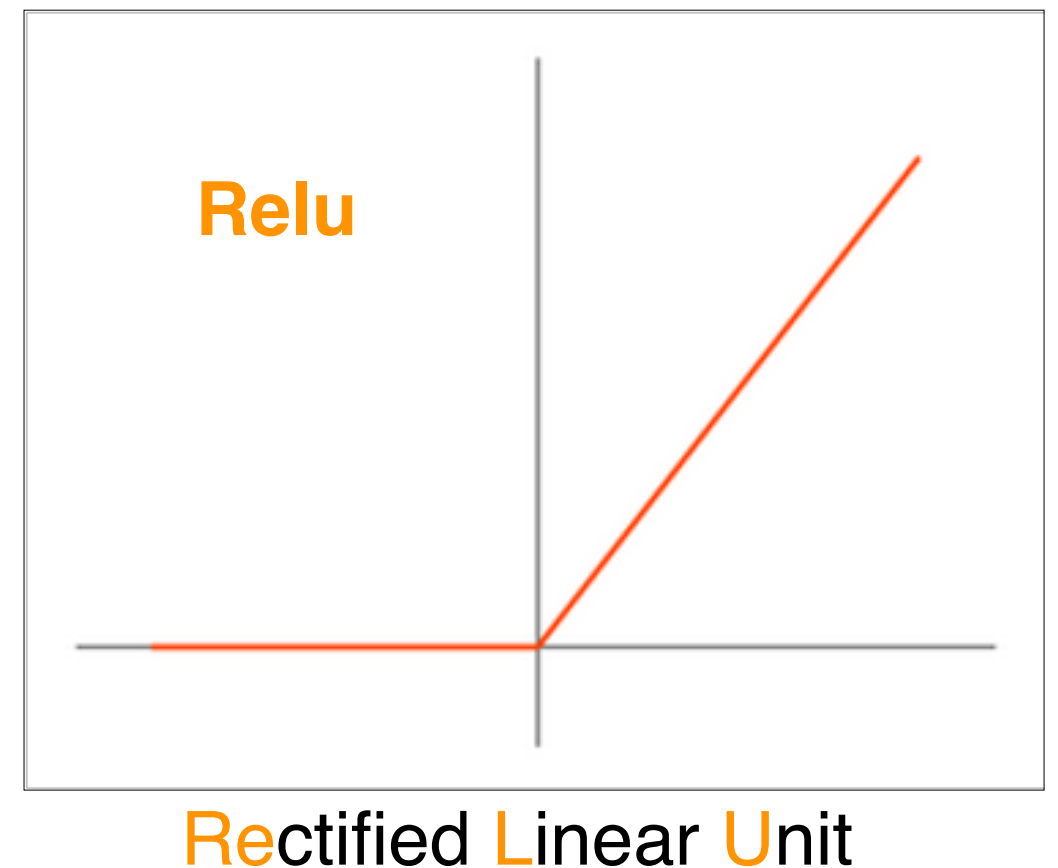
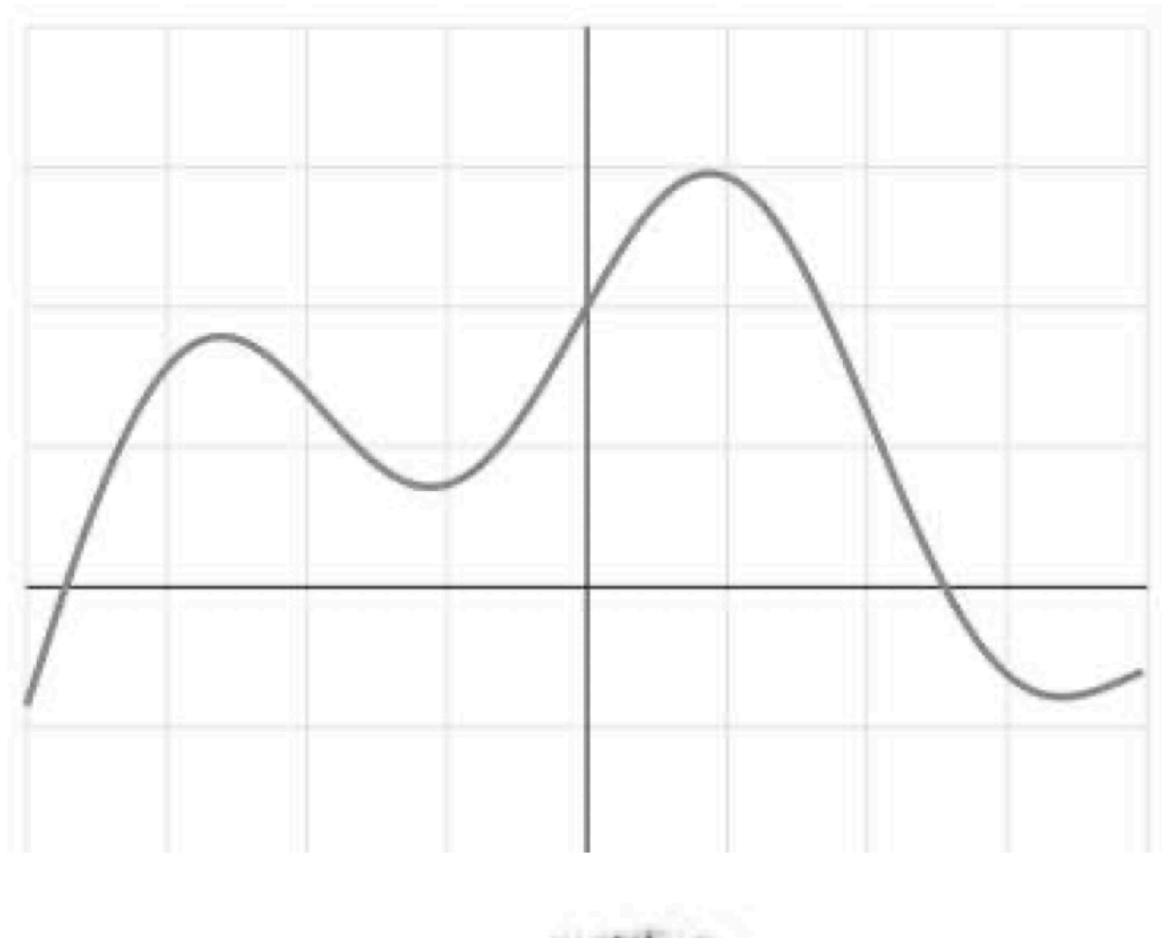


- Returns only **Boolean** inputs (0,1)
- Assembly of neurons: **Turing complete**

The formal neuron: generalisation

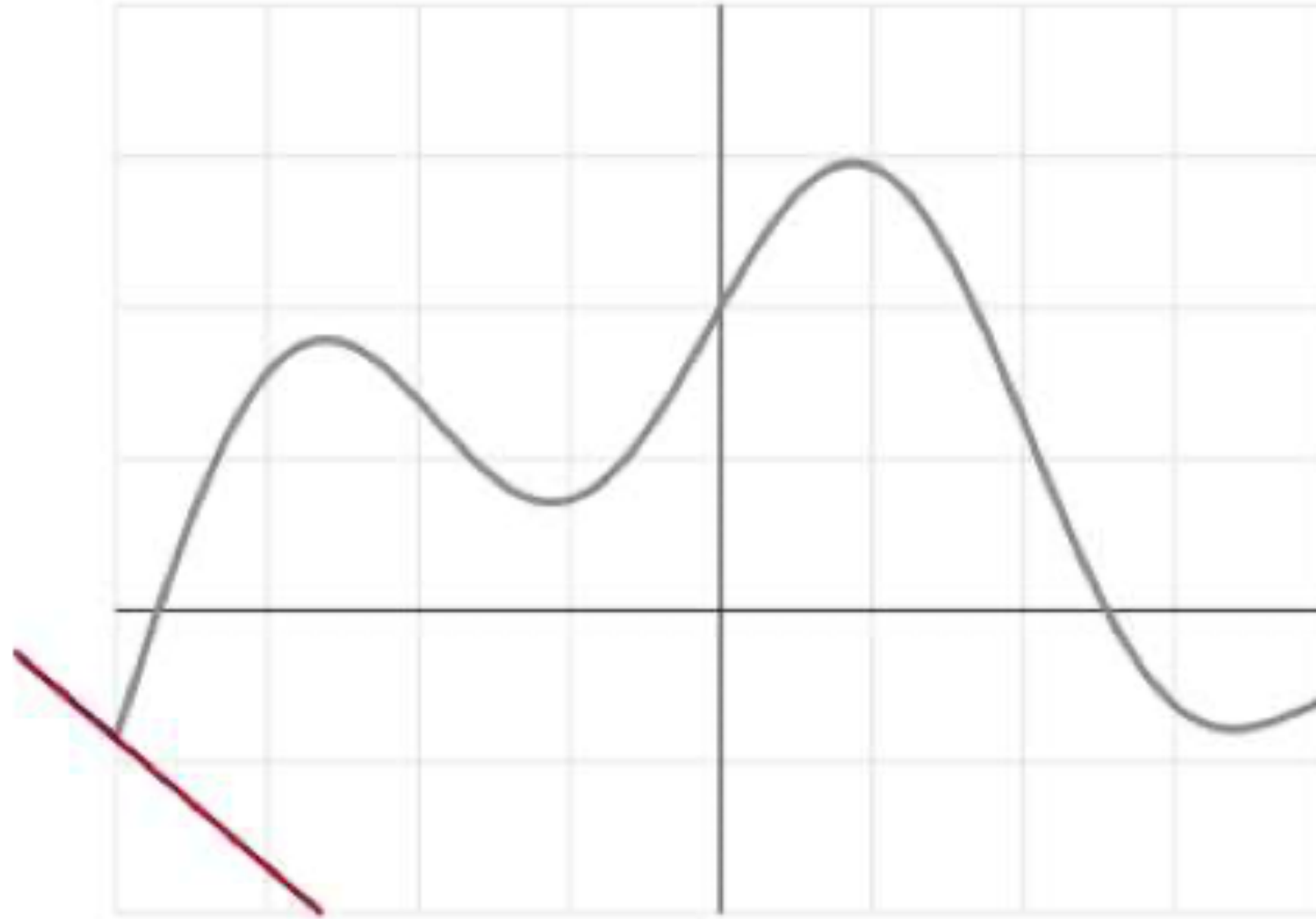


The formal neuron: a universal approximator



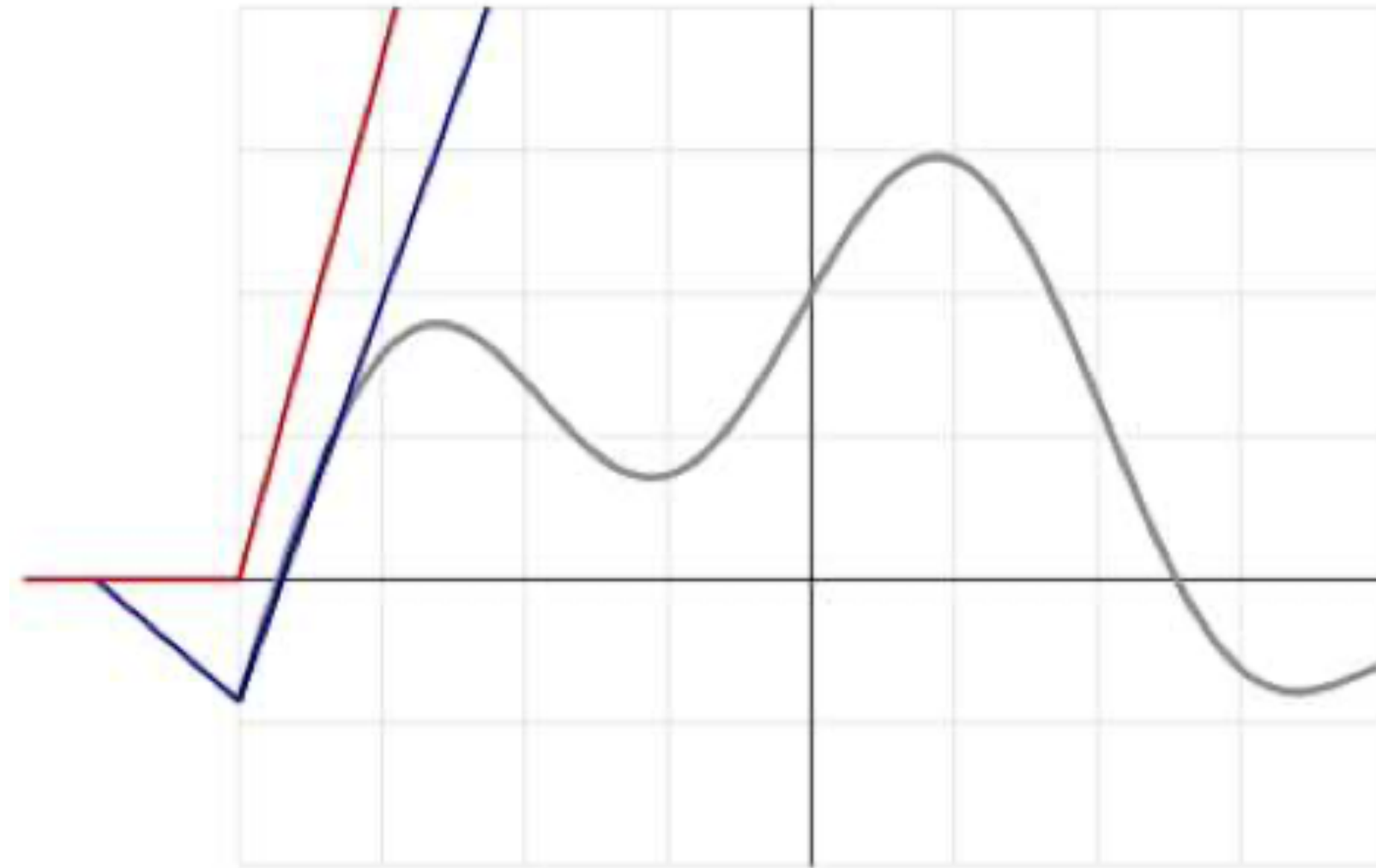
- Any mathematical function operating on Real number can be **approximated** as a sum of translated **ReLU functions**
- Neural networks can naturally fulfil this role

The formal neuron: a universal approximator



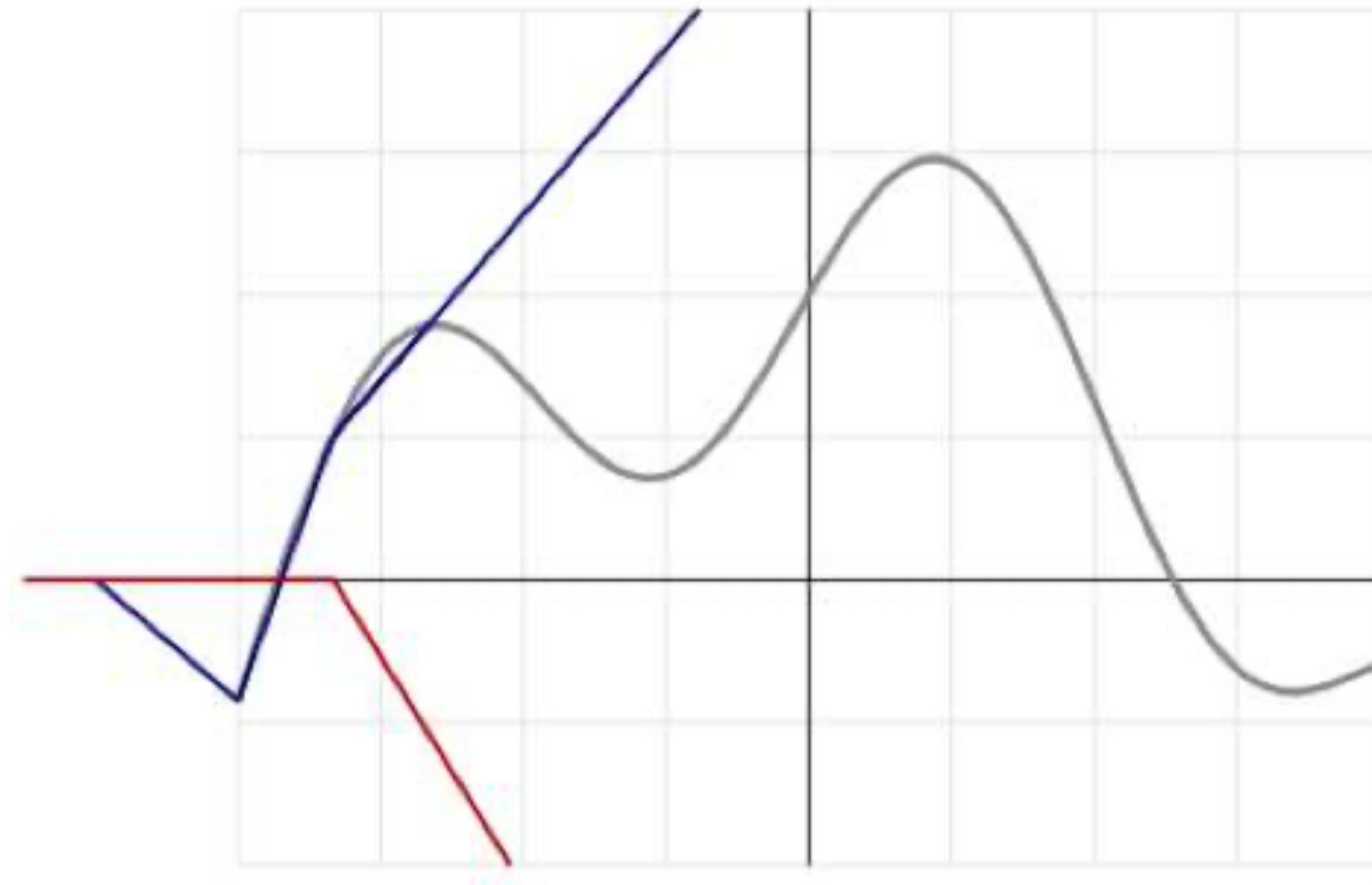
- Any mathematical function operating on Real number can be **approximated** as a sum of translated **ReLU functions**
- Neural networks can naturally fulfil this role

The formal neuron: a universal approximator



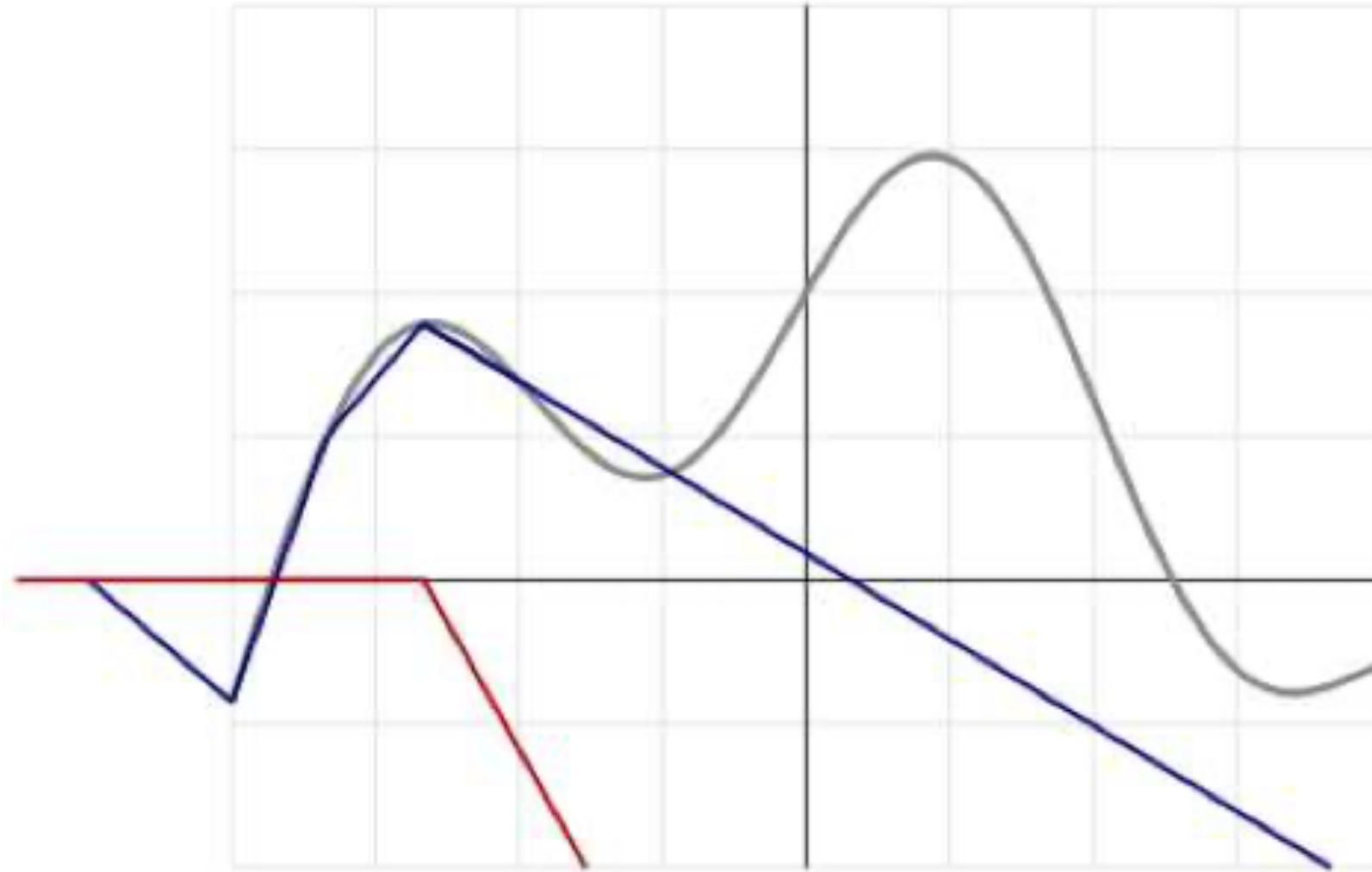
- Any mathematical function operating on Real number can be **approximated** as a sum of translated **ReLU functions**
- Neural networks can naturally fulfil this role

The formal neuron: a universal approximator



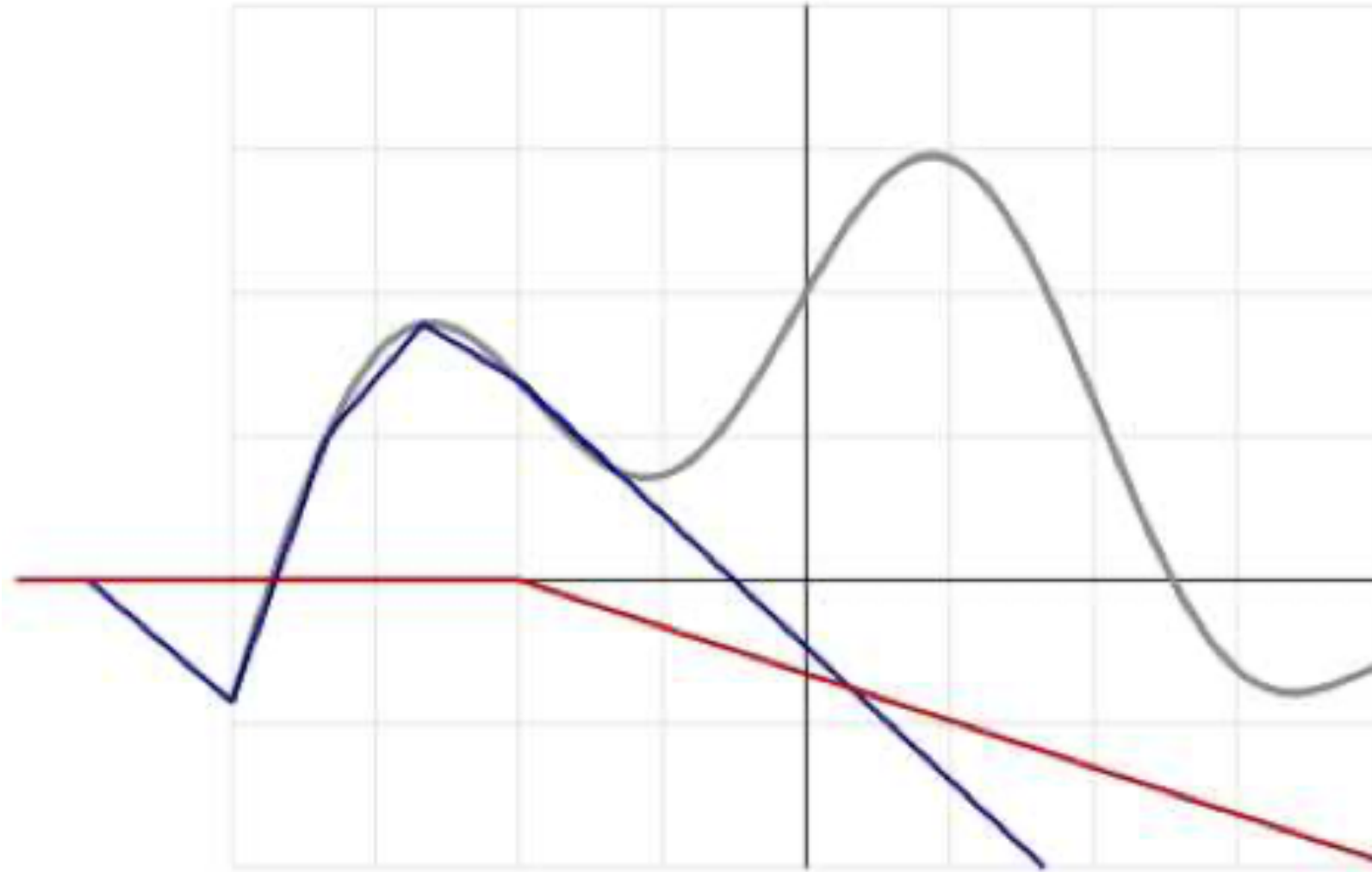
- Any mathematical function operating on Real number can be **approximated** as a sum of translated **ReLU functions**
- Neural networks can naturally fulfil this role

The formal neuron: a universal approximator



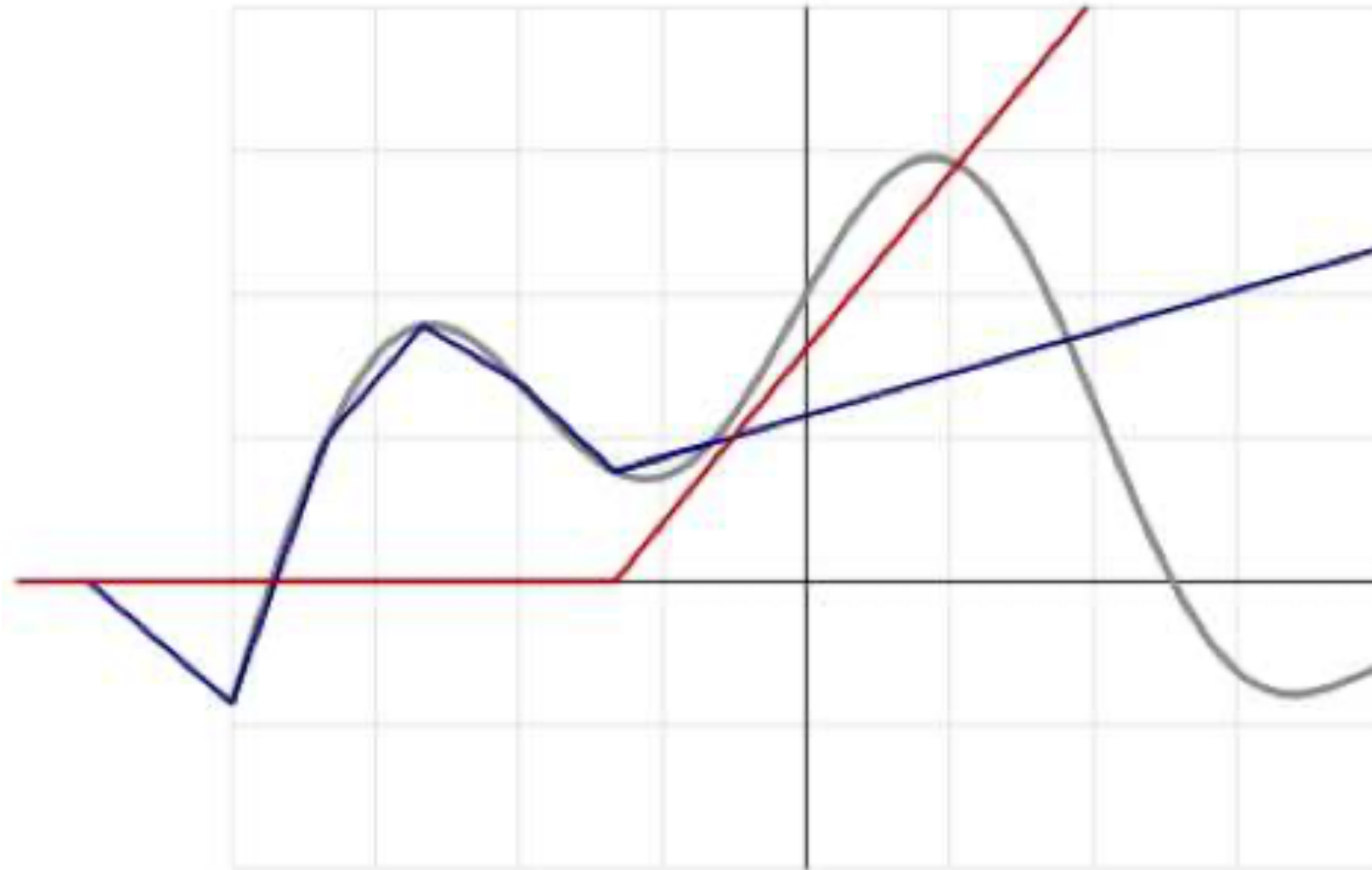
- Any mathematical function operating on Real number can be **approximated** as a sum of translated **ReLU functions**
- Neural networks can naturally fulfil this role

The formal neuron: a universal approximator



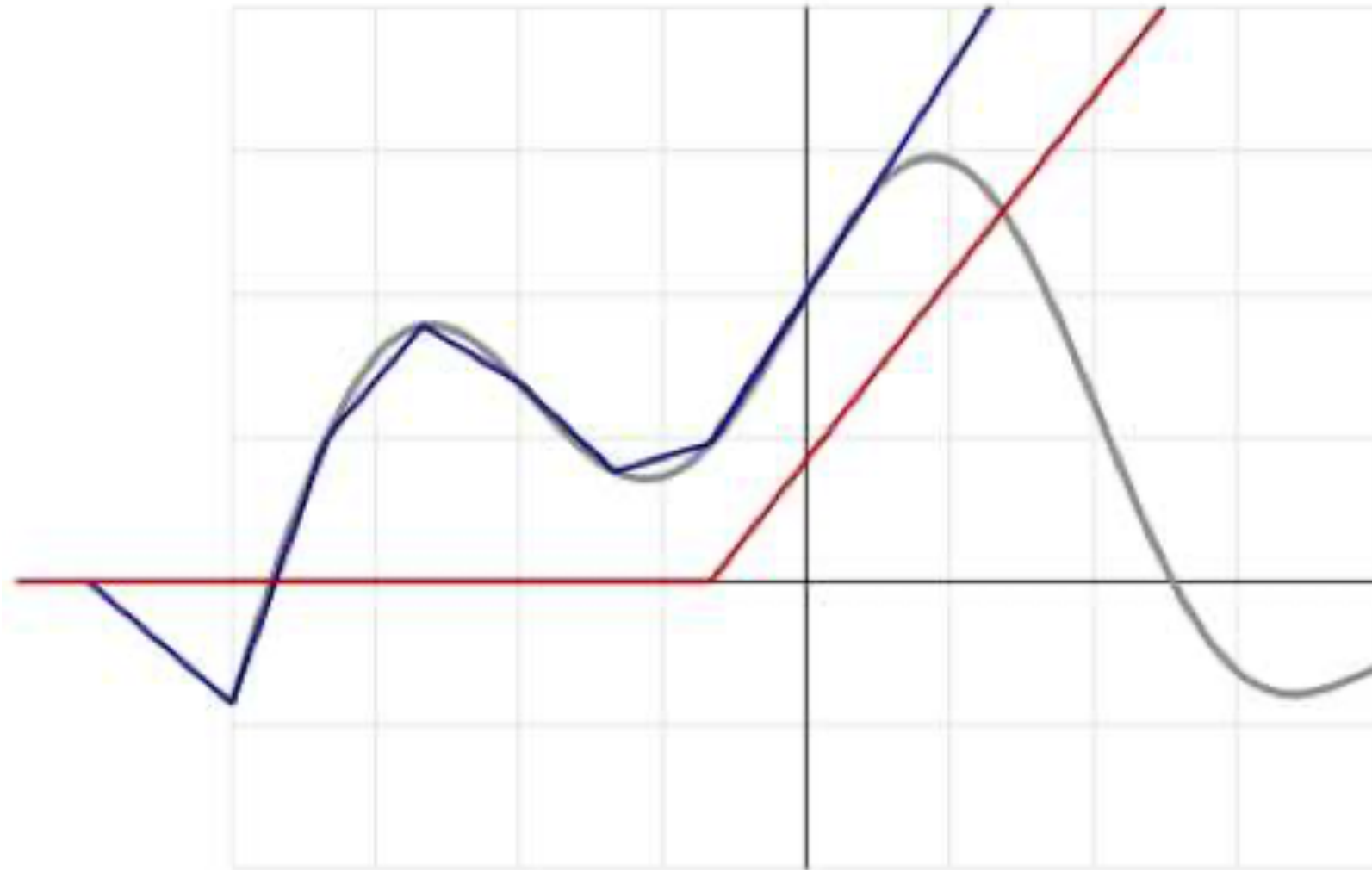
- Any mathematical function operating on Real number can be **approximated** as a sum of translated **ReLU functions**
- Neural networks can naturally fulfil this role

The formal neuron: a universal approximator



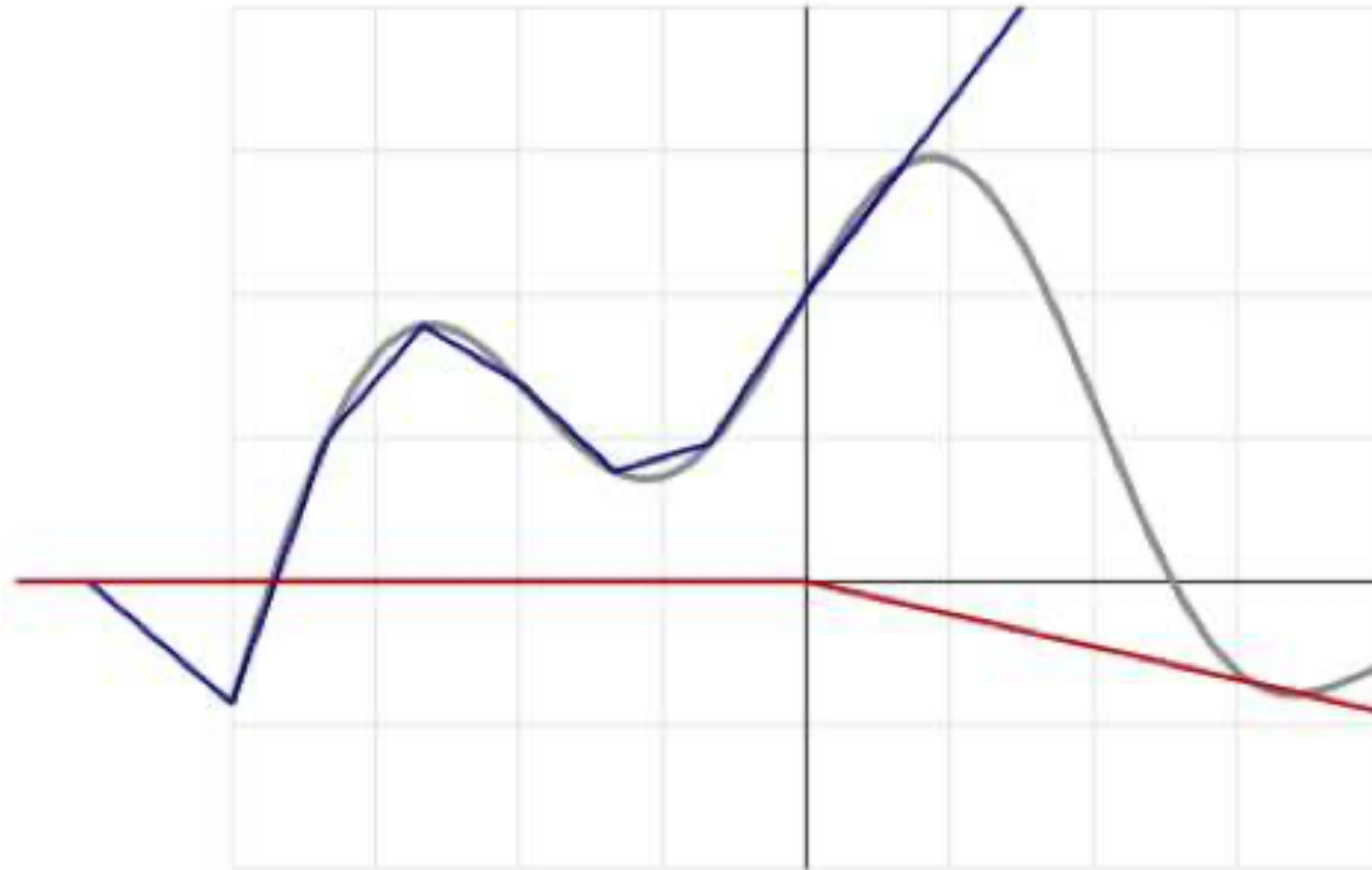
- Any mathematical function operating on Real number can be **approximated** as a sum of translated **ReLU functions**
- Neural networks can naturally fulfil this role

The formal neuron: a universal approximator



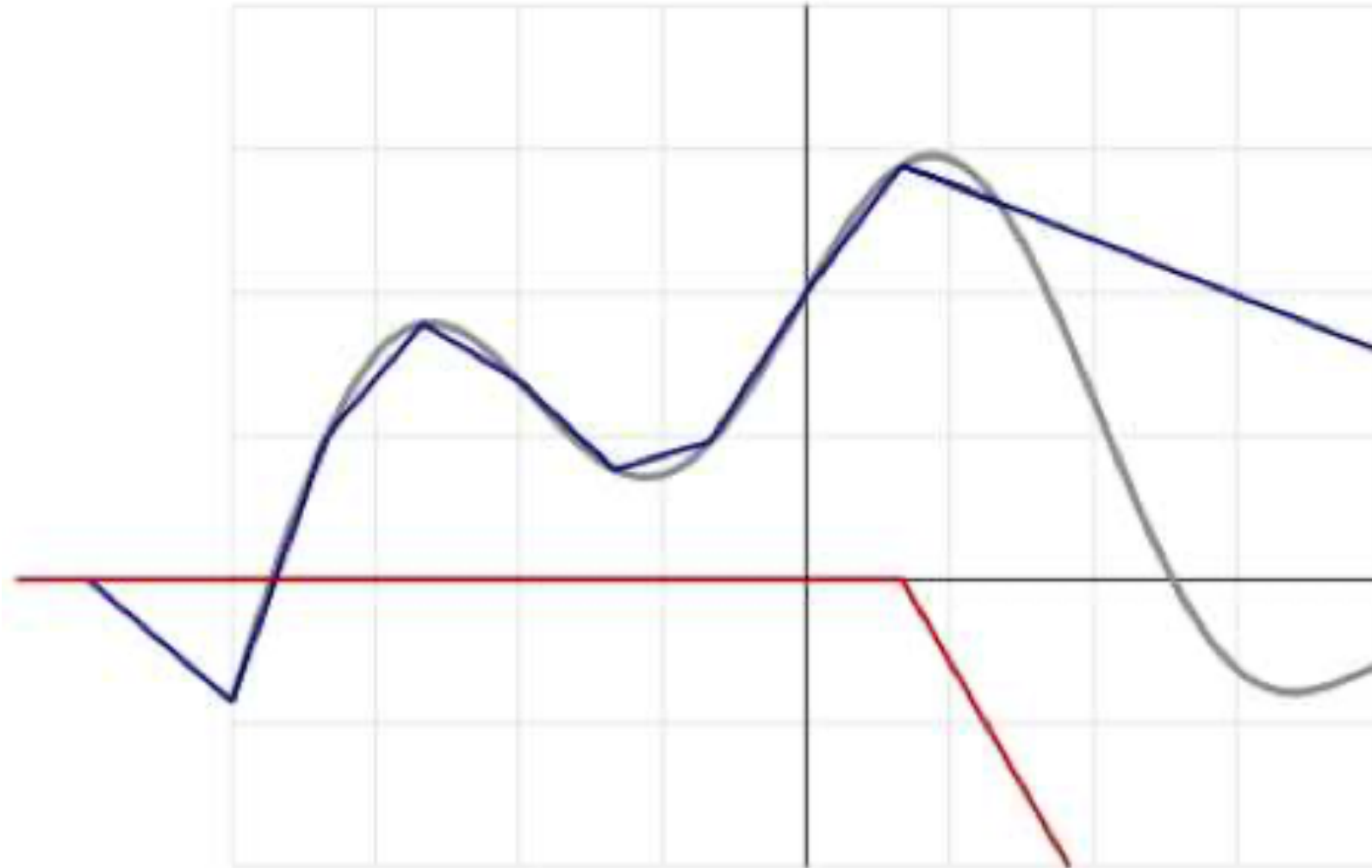
- Any mathematical function operating on Real number can be **approximated** as a sum of translated **ReLU functions**
- Neural networks can naturally fulfil this role

The formal neuron: a universal approximator



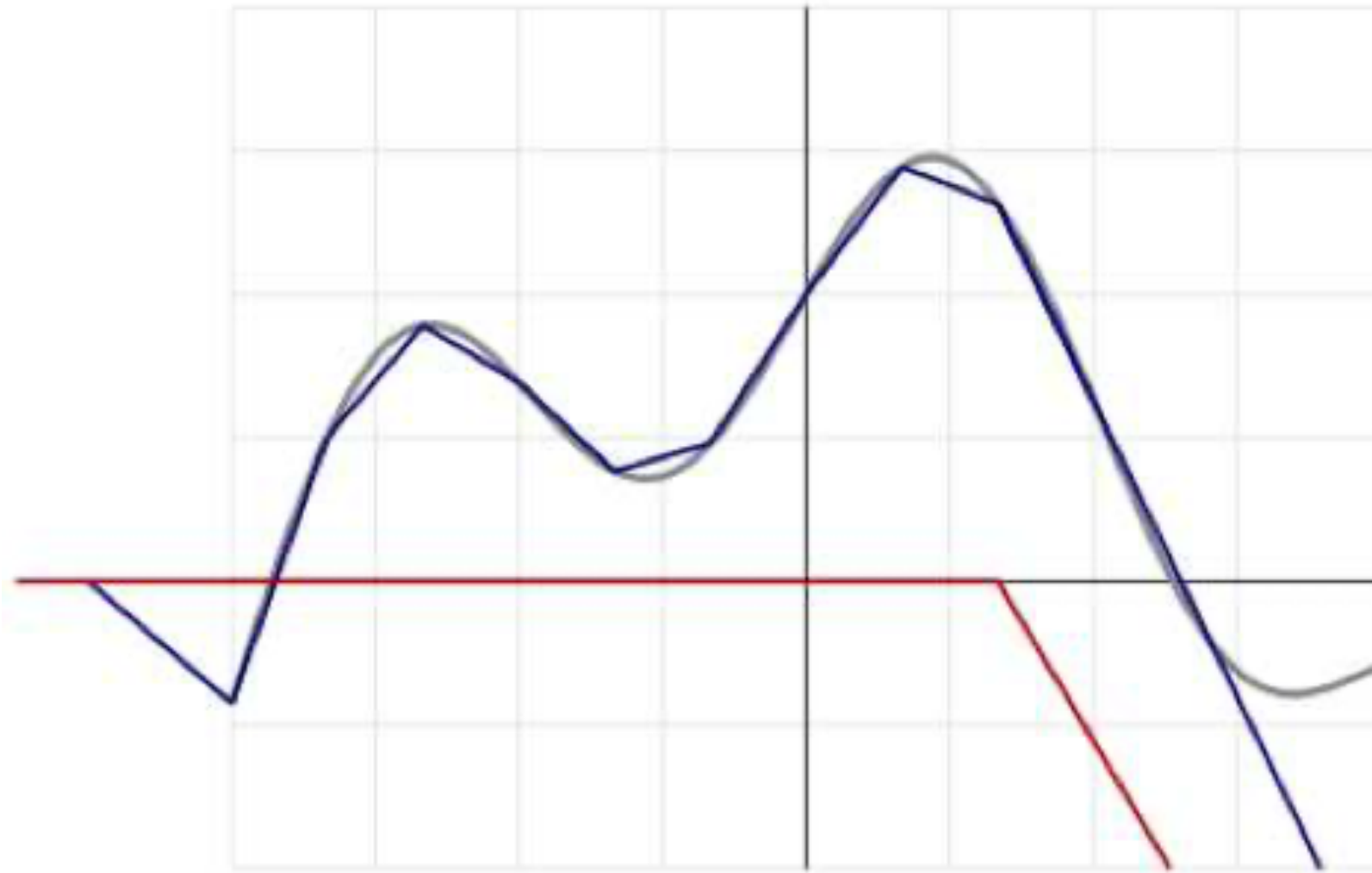
- Any mathematical function operating on Real number can be **approximated** as a sum of translated **ReLU functions**
- Neural networks can naturally fulfil this role

The formal neuron: a universal approximator



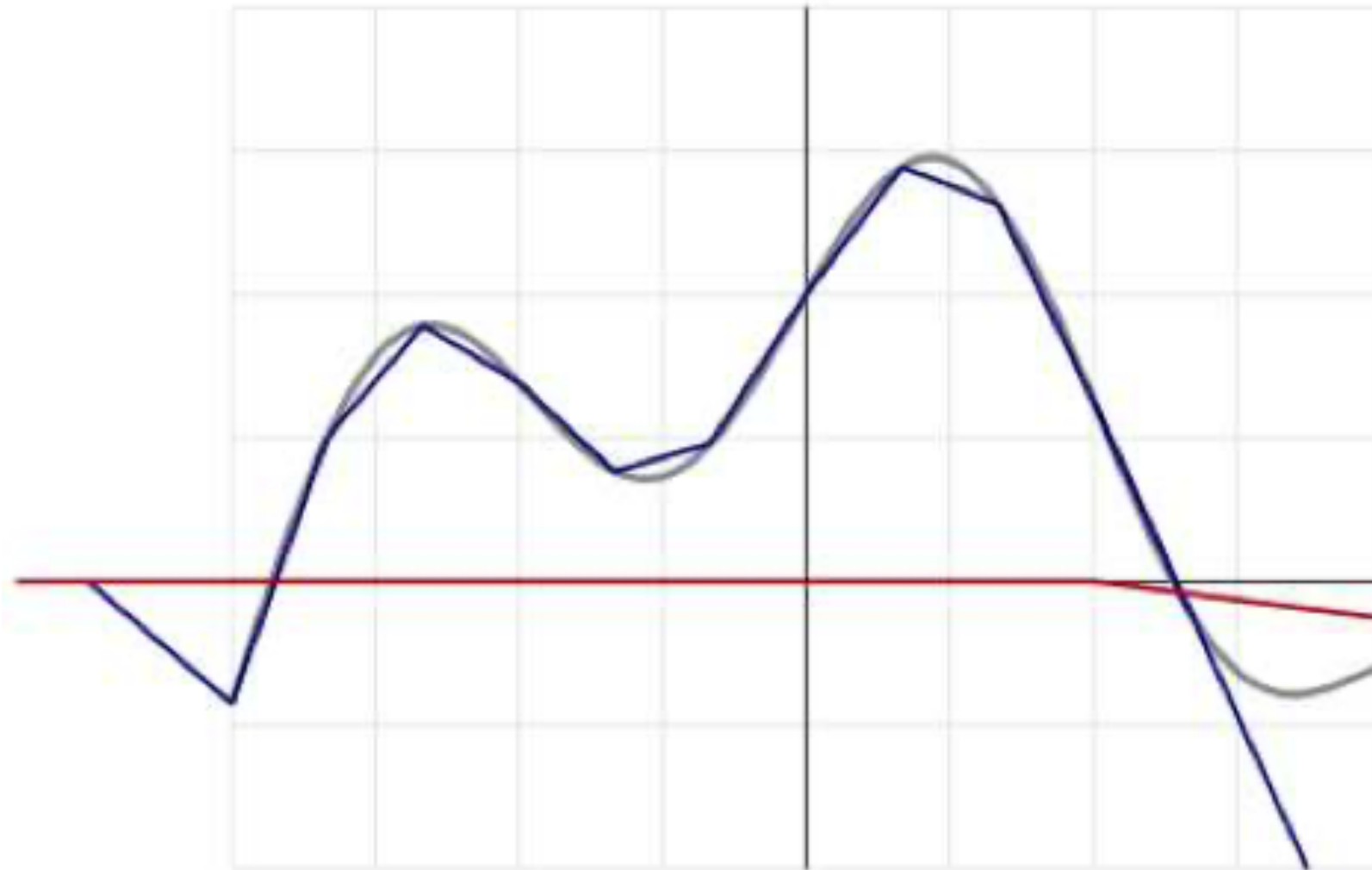
- Any mathematical function operating on Real number can be **approximated** as a sum of translated **ReLU functions**
- Neural networks can naturally fulfil this role

The formal neuron: a universal approximator



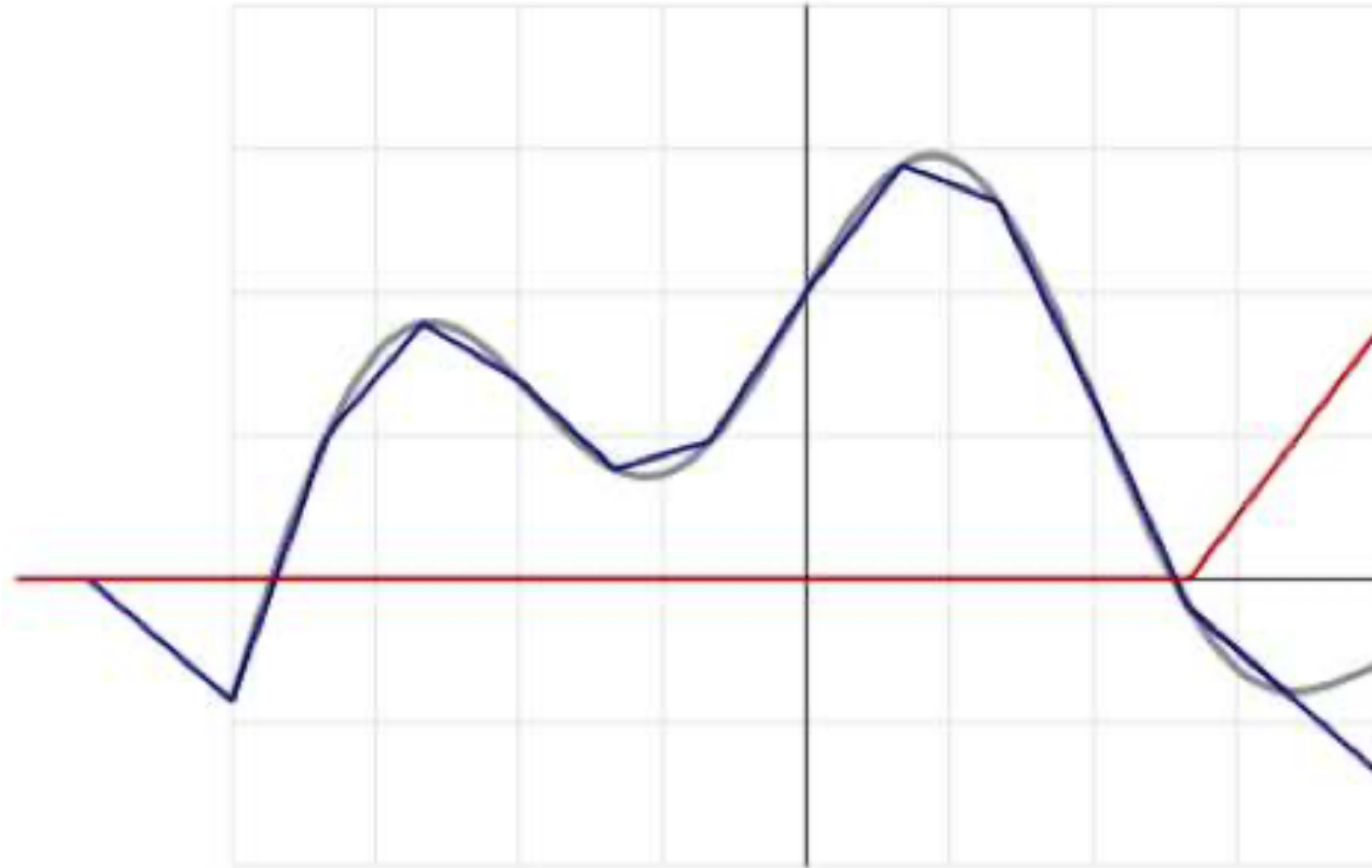
- Any mathematical function operating on Real number can be **approximated** as a sum of translated **ReLU functions**
- Neural networks can naturally fulfil this role

The formal neuron: a universal approximator



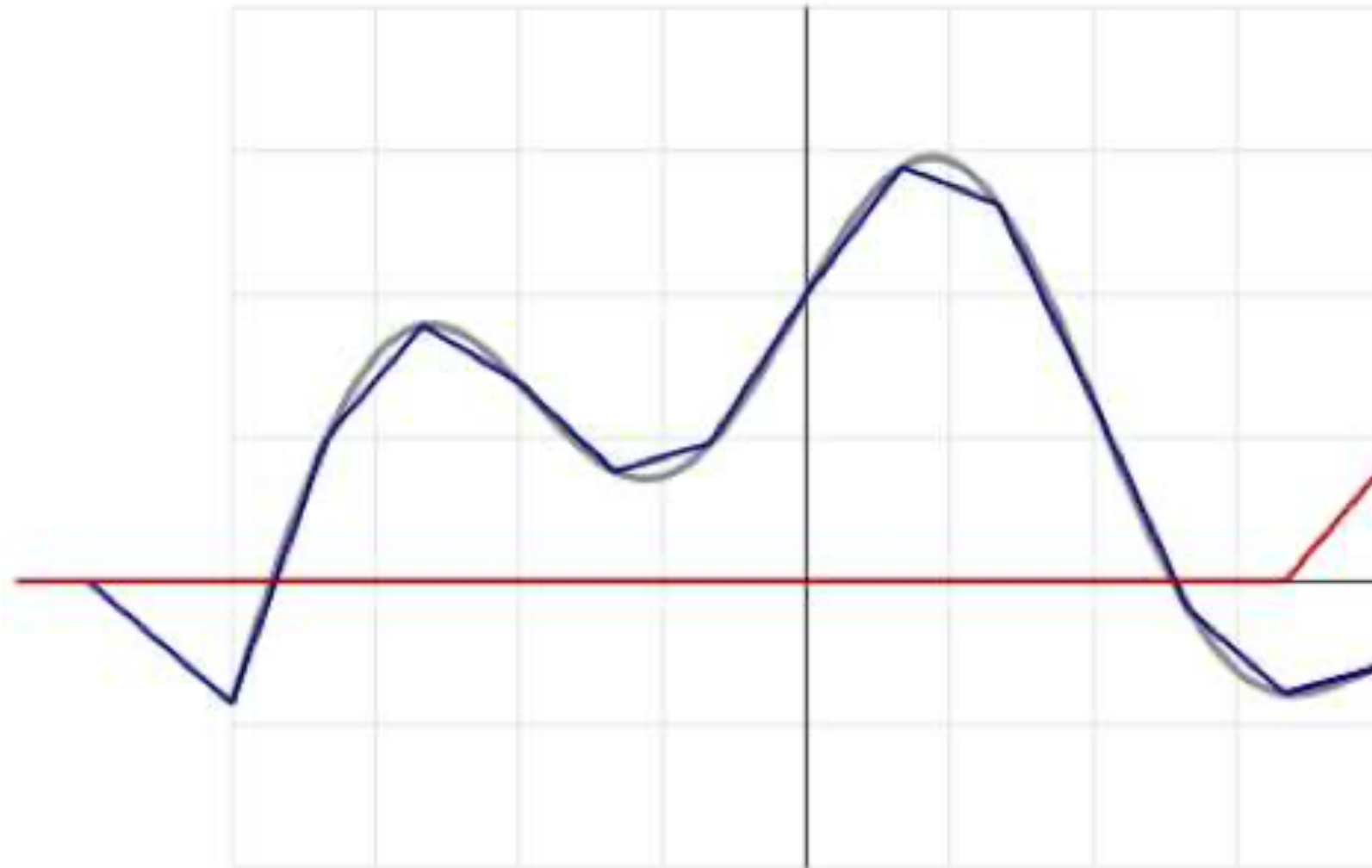
- Any mathematical function operating on Real number can be **approximated** as a sum of translated **ReLU functions**
- Neural networks can naturally fulfil this role

The formal neuron: a universal approximator



- Any mathematical function operating on Real number can be **approximated** as a sum of translated **ReLU functions**
- Neural networks can naturally fulfil this role

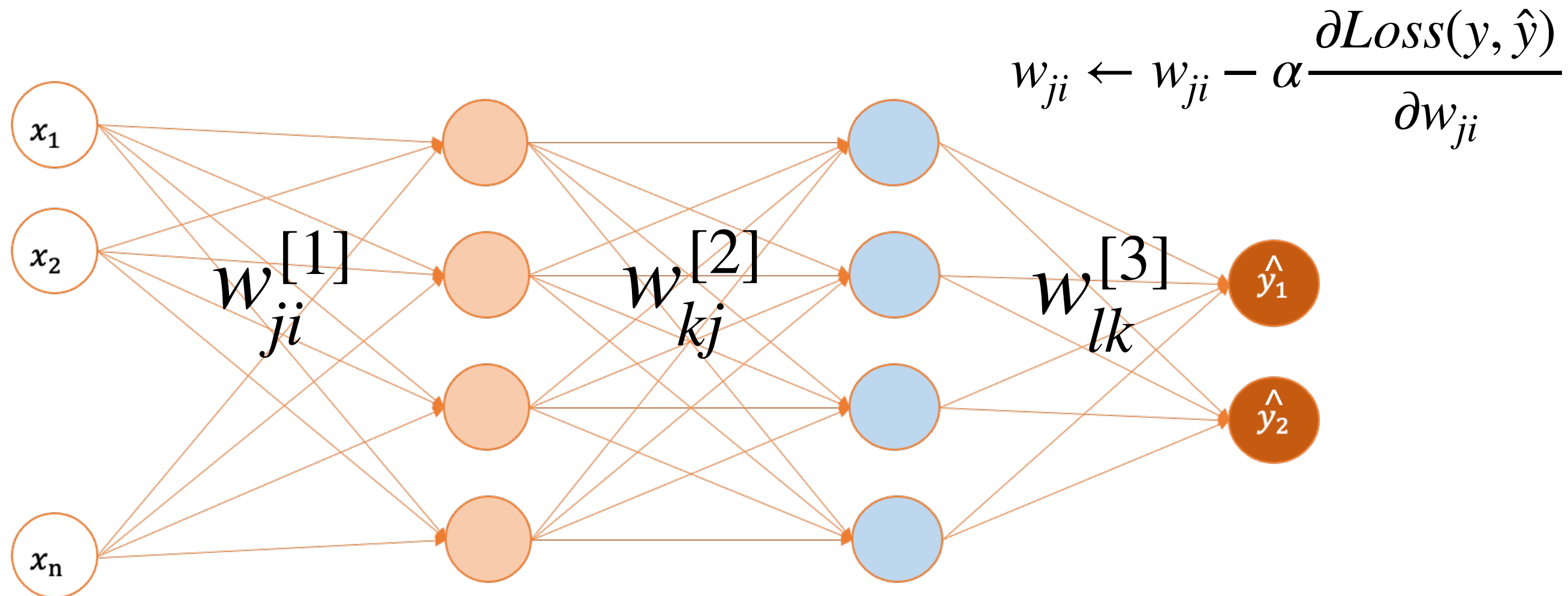
The formal neuron: a universal approximator



- Any mathematical function operating on Real number can be **approximated** as a sum of translated **ReLU functions**
- Neural networks can naturally fulfil this role

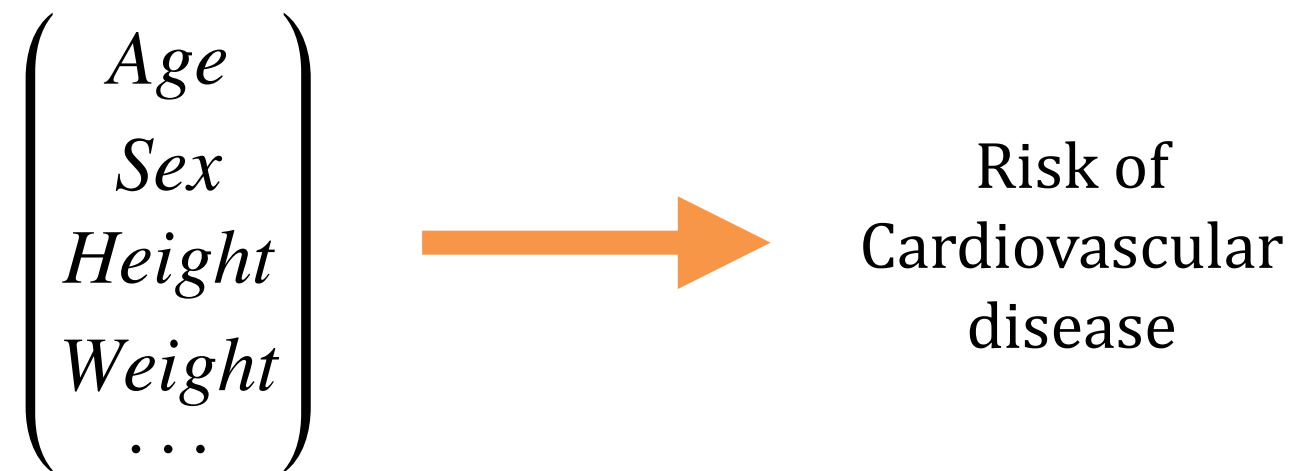
The neural network

- Neurons combined into networks:
the **output** of layer N-1 is used as **input** for layer N
- **Training:** tuning all weights W such as $\hat{y} = y$
- Weight updated using the **partial derivative** of a loss function



Limits of neural networks (for HEP)

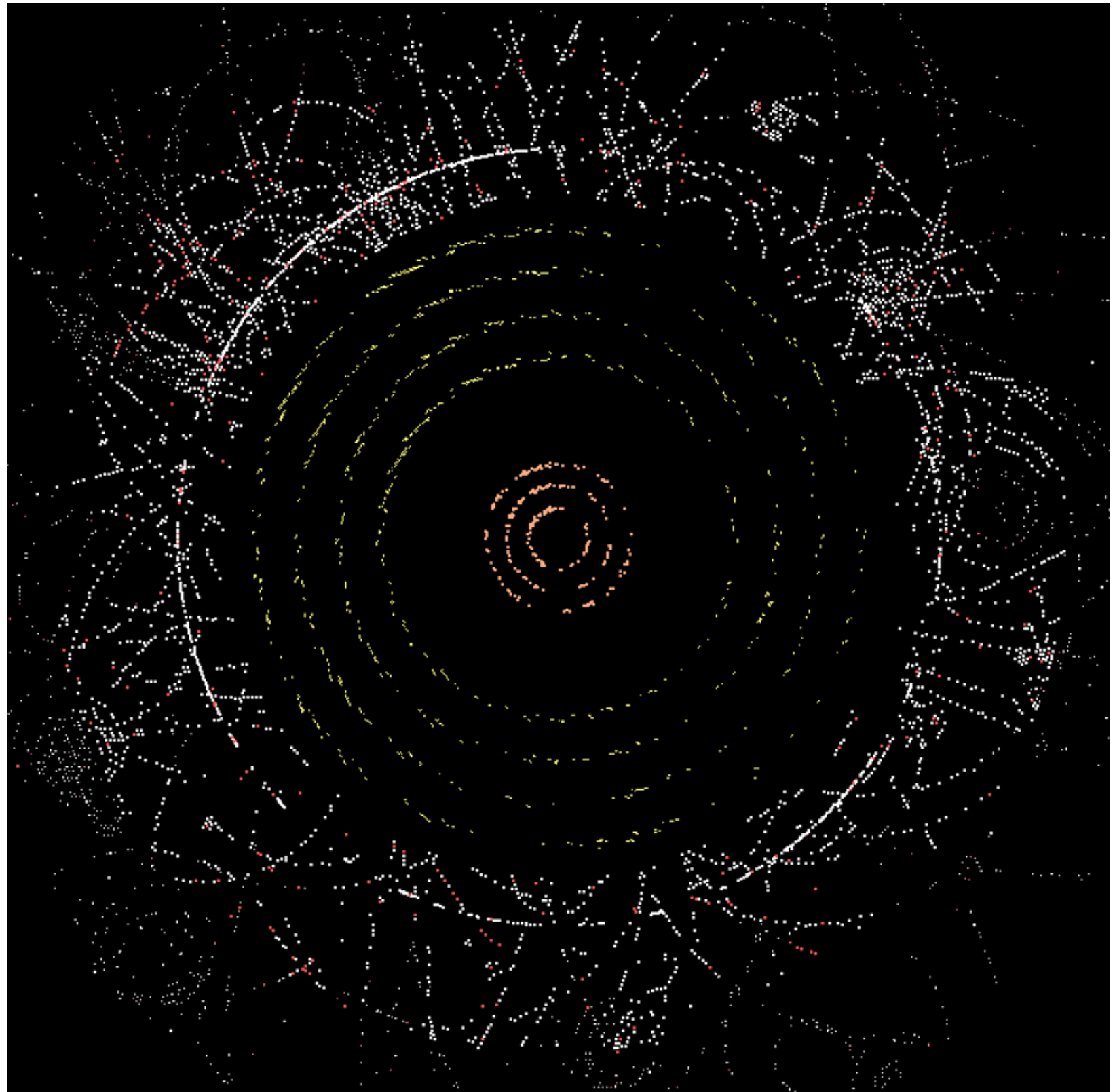
- Neural networks operate on ordered tabular data
 - ➡ Great when making predictions about an **object with fixed attributes**



- How to handle input with multiple objects? (with their own attributes)
 - ➡ No guarantee of **permutation invariance**... (some trick exist)
 - ➡ Use an arbitrary ordering? What happens if we order incorrectly ?
 - ➡ Reconstruct known combined quantities (what we do)

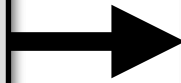
Particle Physics

**Detector
Data
Collection**

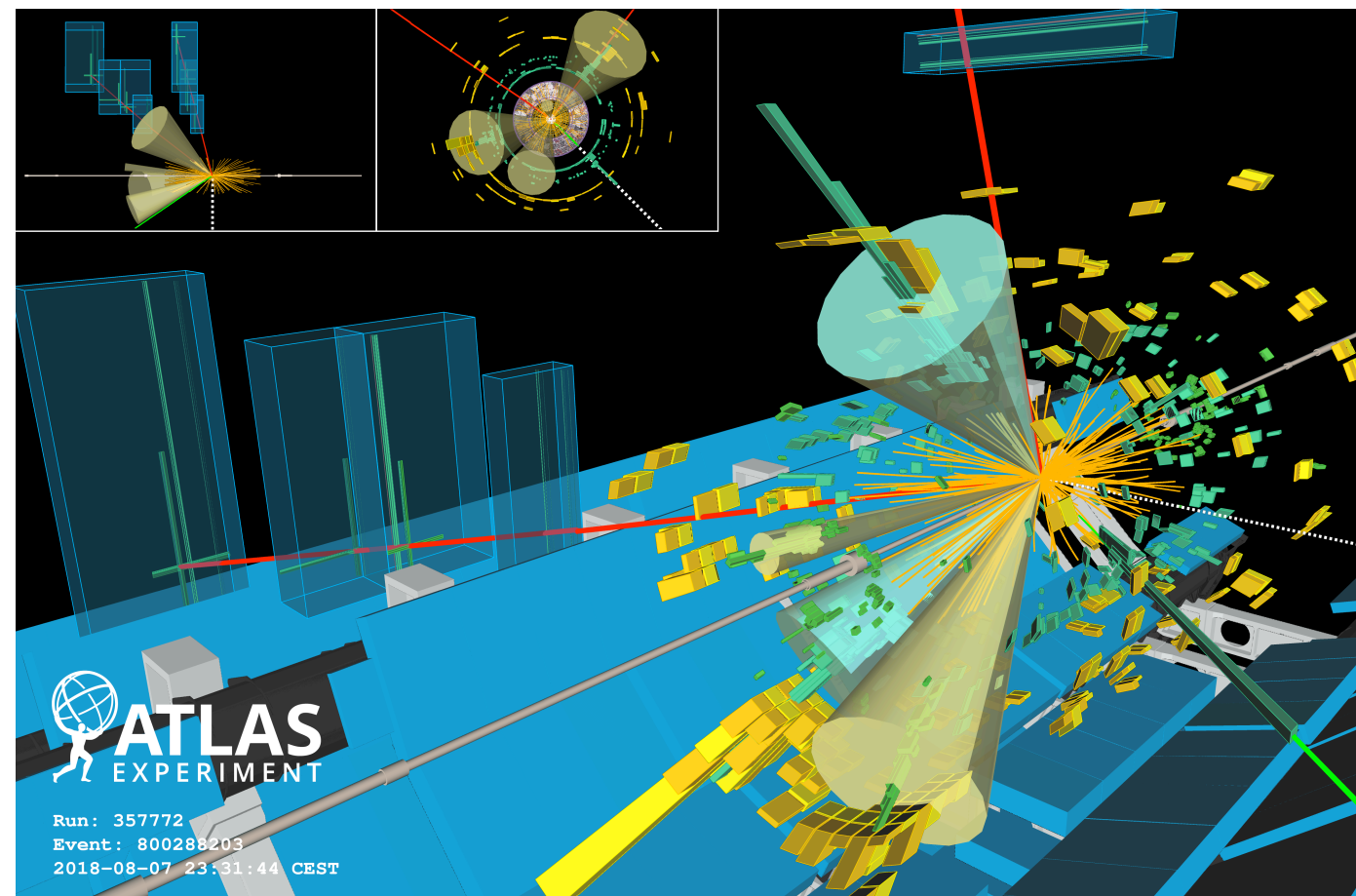
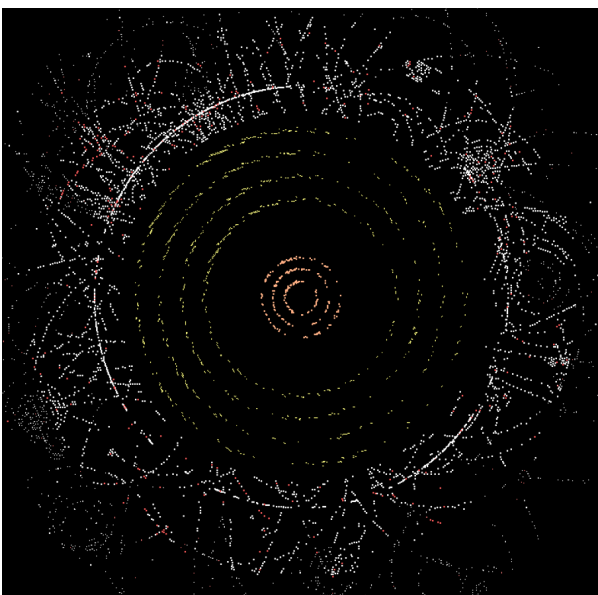


Particle Physics

Detector
Data
Collection



Detector
Object
Processing

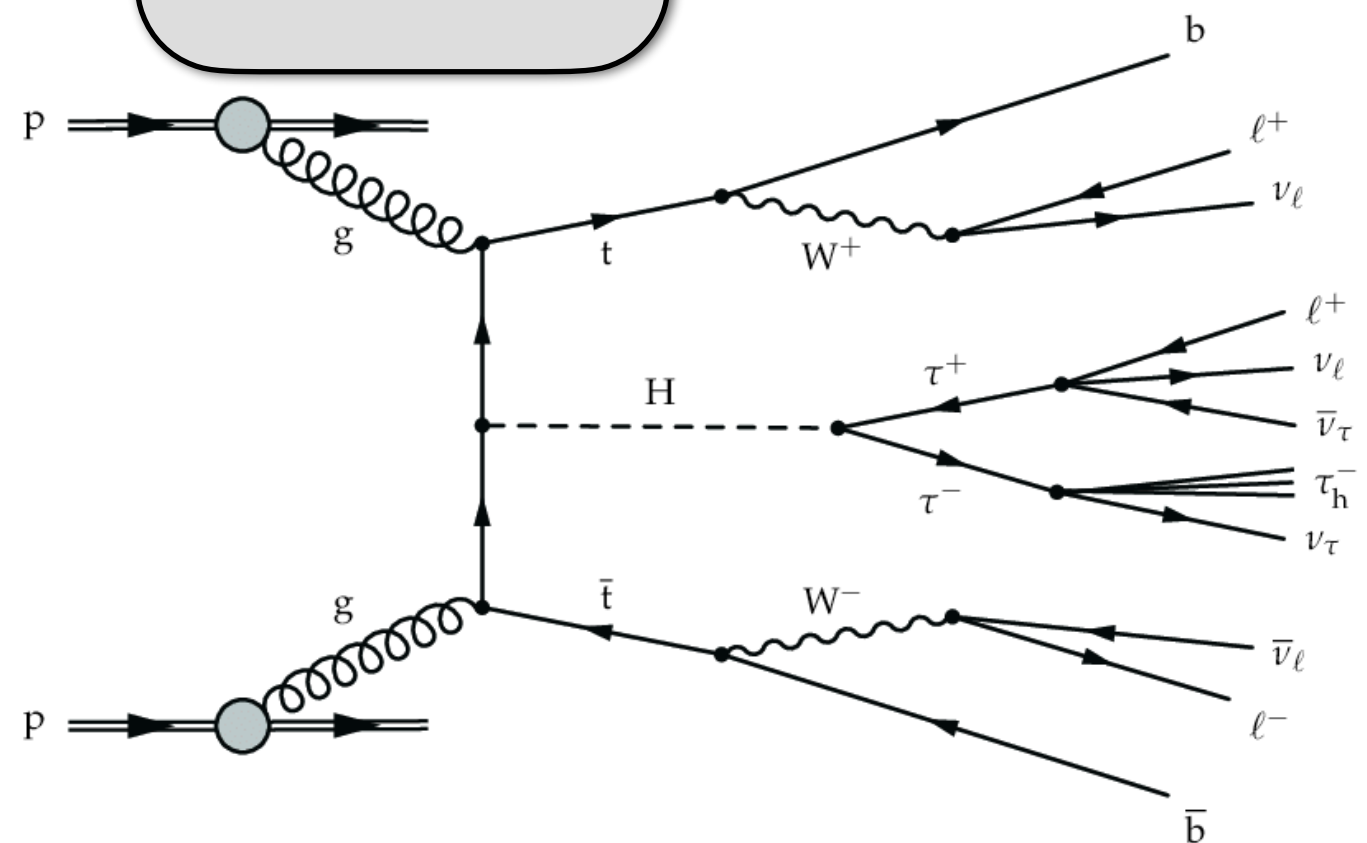
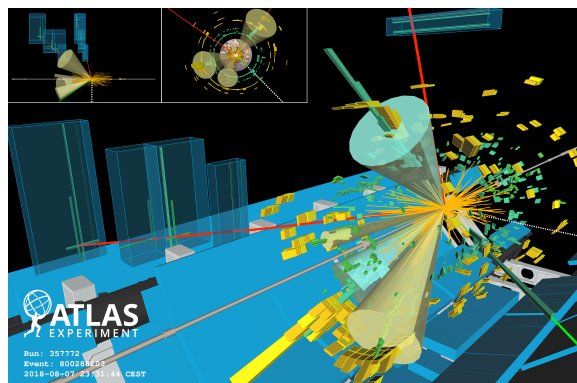
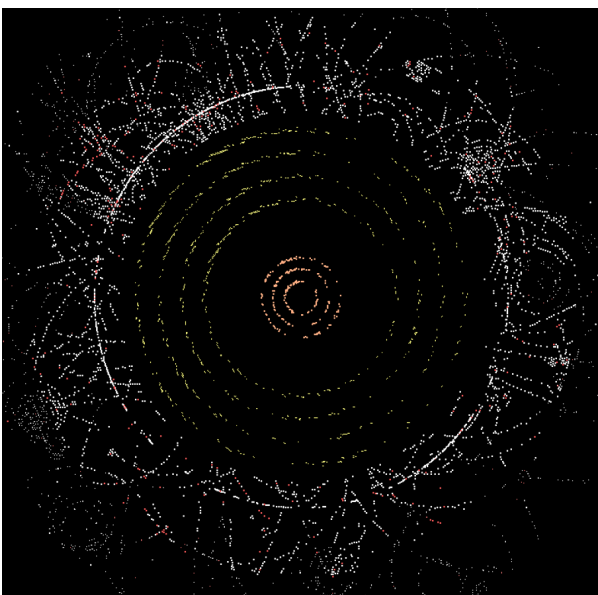


Particle Physics

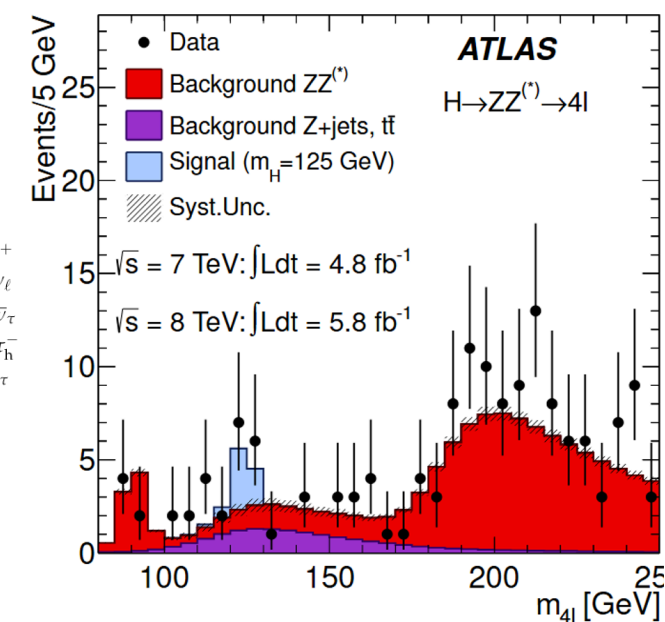
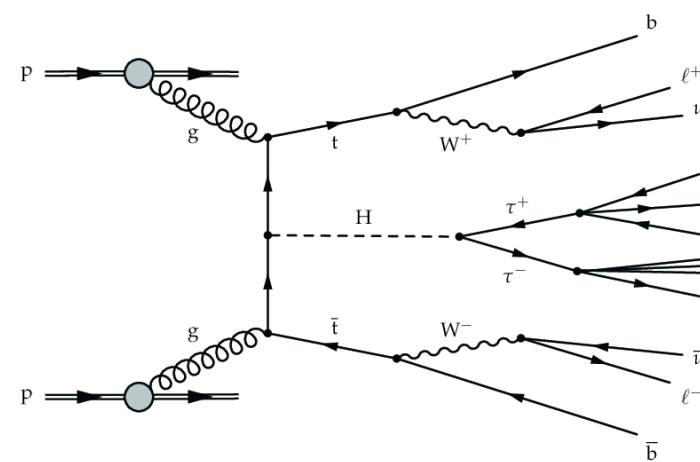
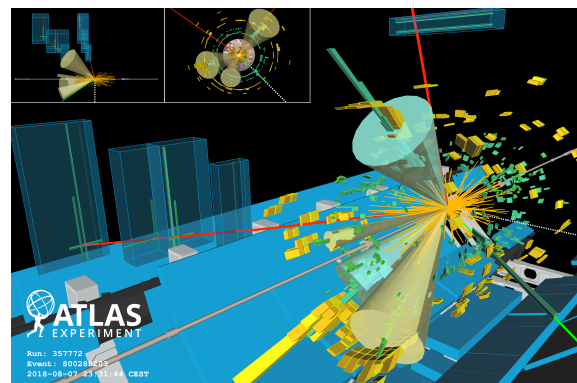
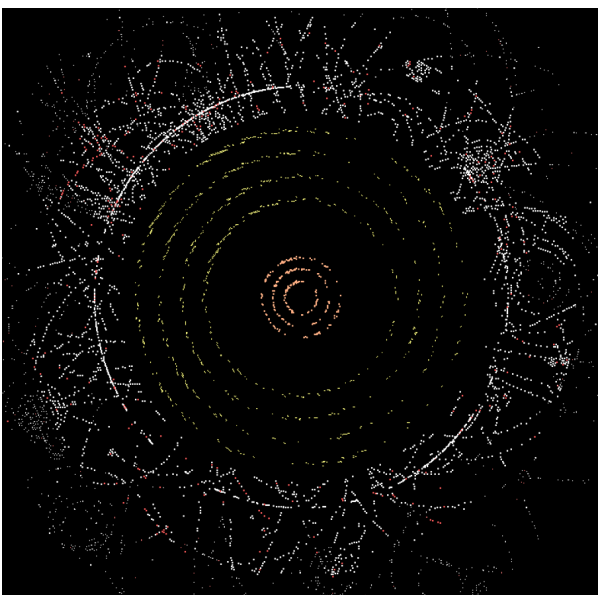
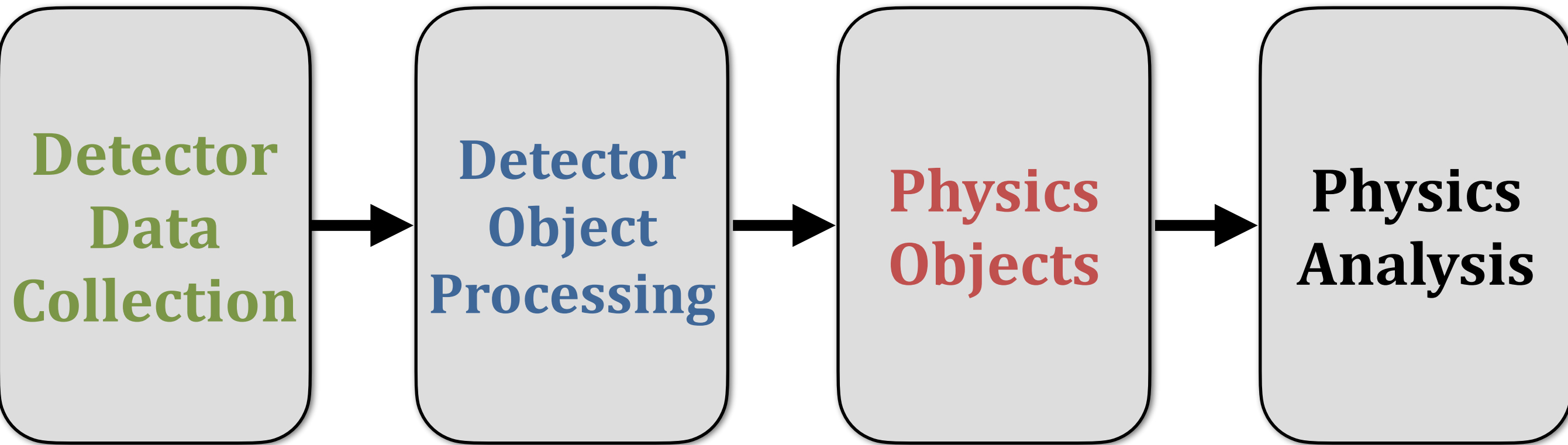
Detector
Data
Collection

Detector
Object
Processing

Physics
Objects

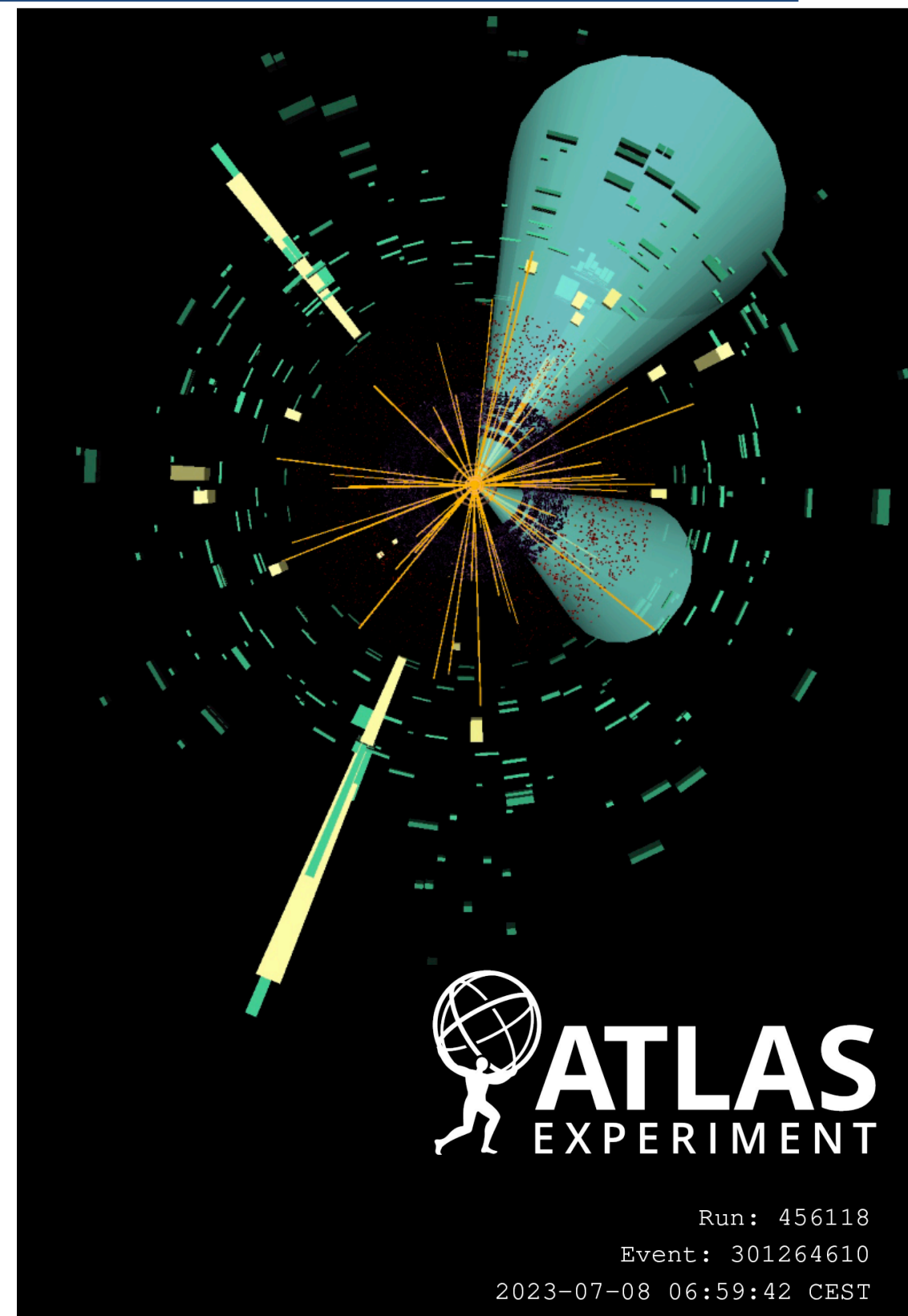


Particle Physics



Limits of neural networks (for HEP)

- HEP data fall in the second category
- Example: $HH \rightarrow b\bar{b}\gamma\gamma$
 - 4 objects: 2 photons, 2 jets
 - How do we order the particle? Leading pT first? Jets then electrons?
- Issue even more present when looking at objects with more subcomponents such as **jets** or **tracks**



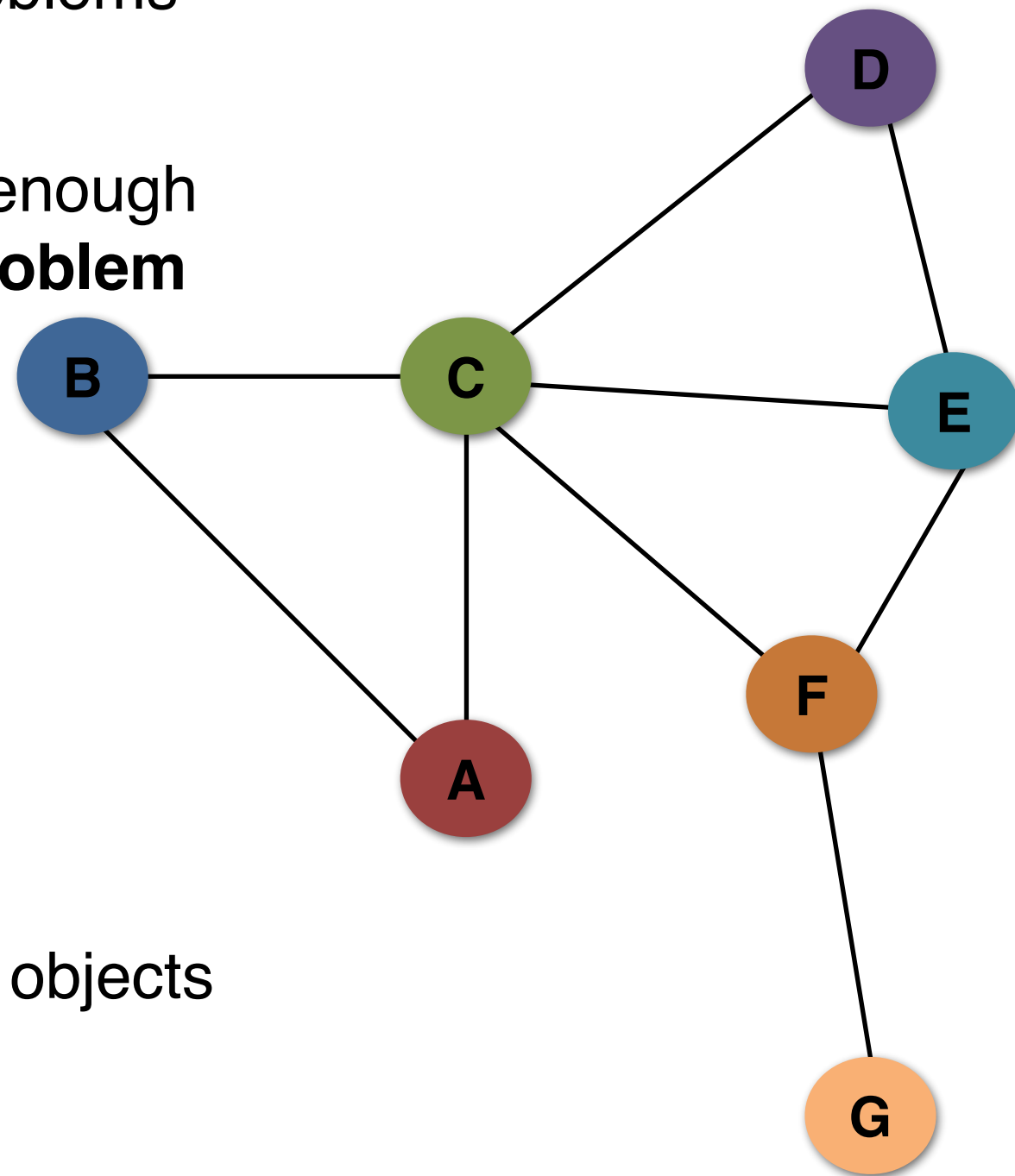
From DNN to GNN

- **Solutions?**

- ➡ Network architecture adapted to our problems
- ➡ Good practice for DL in general:
try the **simplest** possible solution; if not enough
try an architecture that **matches your problem**

- **GNN (Graph Neural Network)**

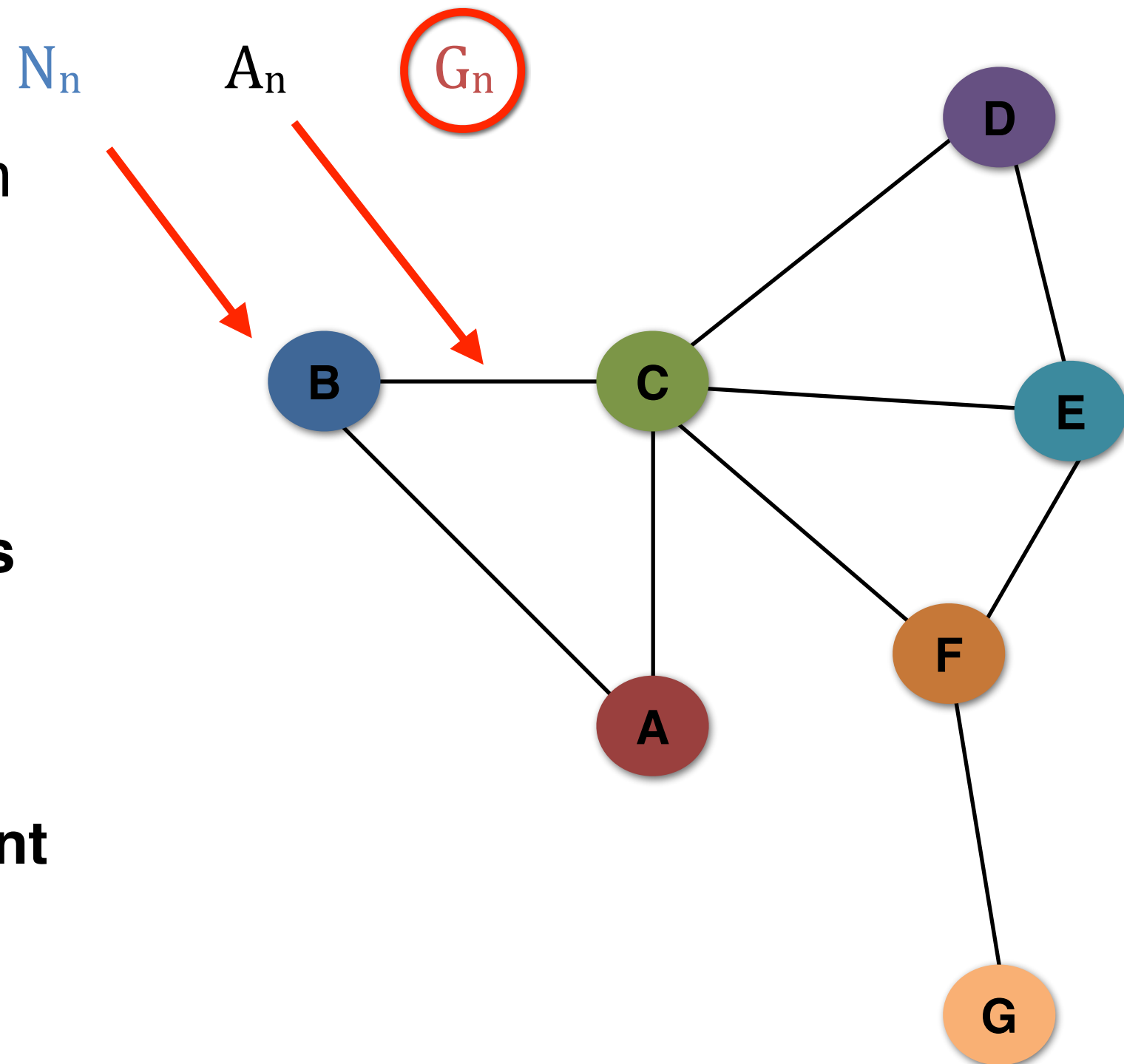
- ➡ Network modified to run on
graph-structured input
- ➡ Each object is now a **node** of the graph
- ➡ Capture the **interaction** between those objects
through **edges**



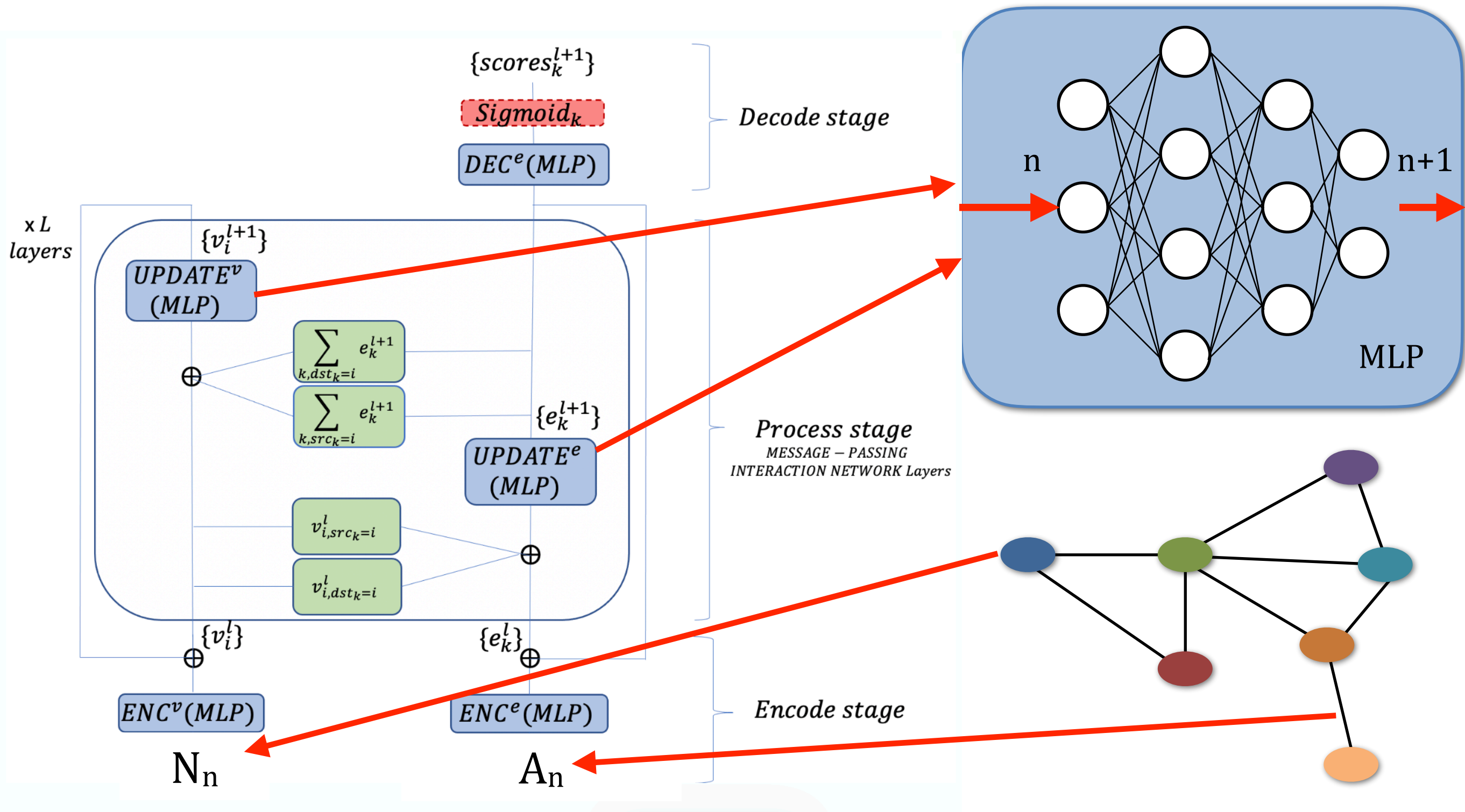
From DNN to GNN

- Implement 3 type of variables : **Node**, Edge and **Global**

- At the **initialisation** you can set any of those to your inputs
- In our **Higgs** case:
 - Two nodes for the **electrons** (and their property)
 - Two nodes for the **Jets**
 - **The network is independent of the graph**: can add extra object without issue

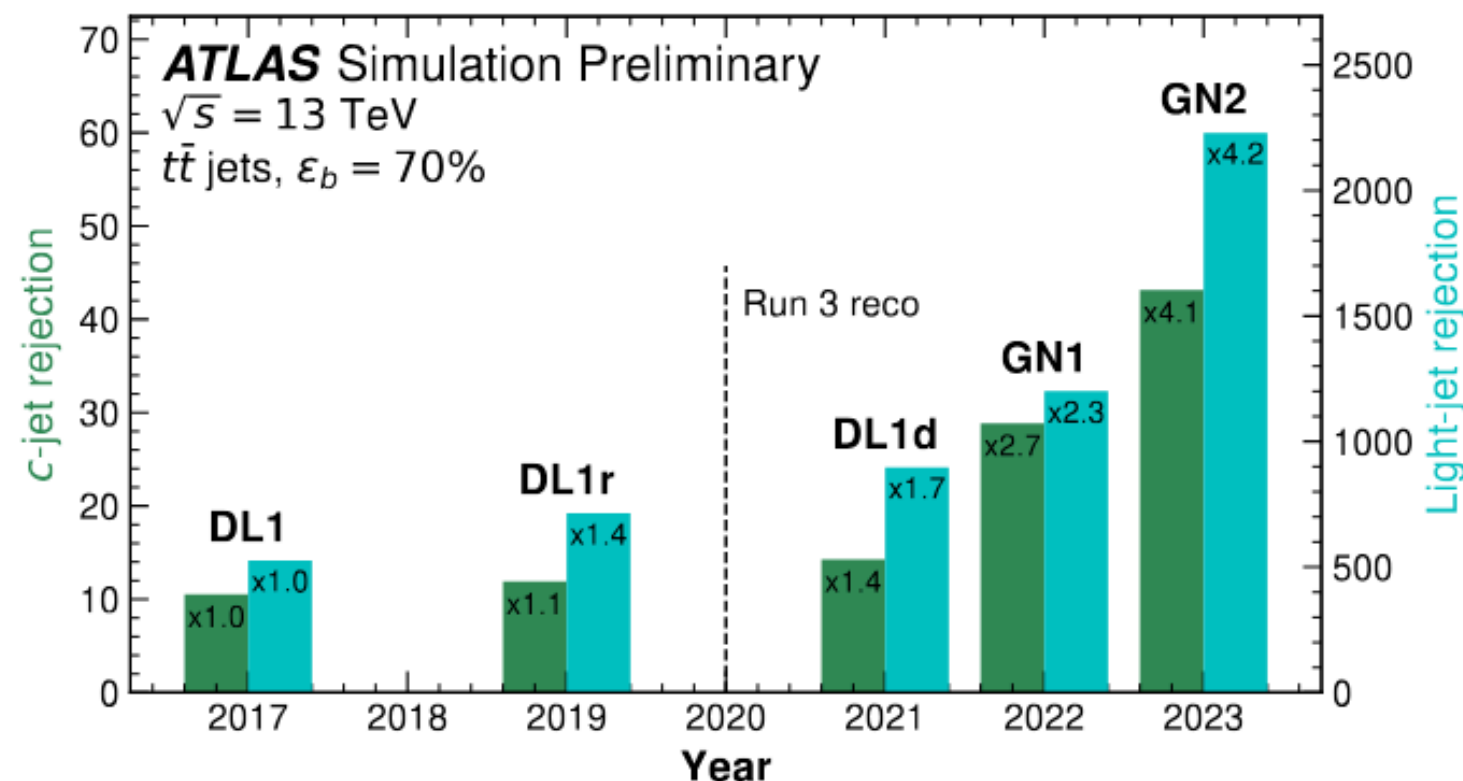
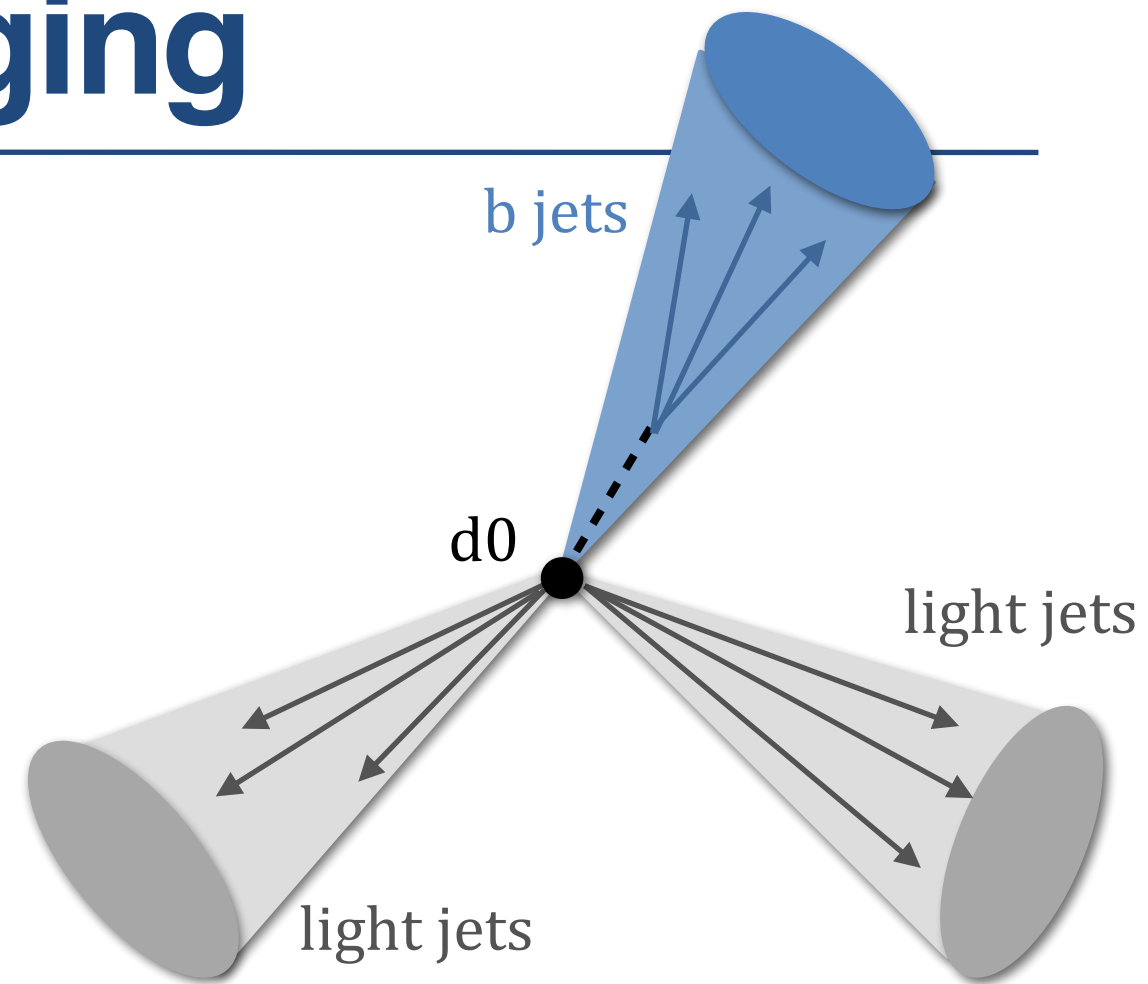


From DNN to GNN



GNN for particle tagging

- Example in ATLAS: jet tagging
- Determine what particle is at the origin of a jet of particles in the detector
- GN1: Extremely good improvement compared to the DL baseline
- Why move to Transformer ?
- **Transformer** can be seen as fully-connected GNN with extra features
- Has been shown to perform extremely well on language-based tasks

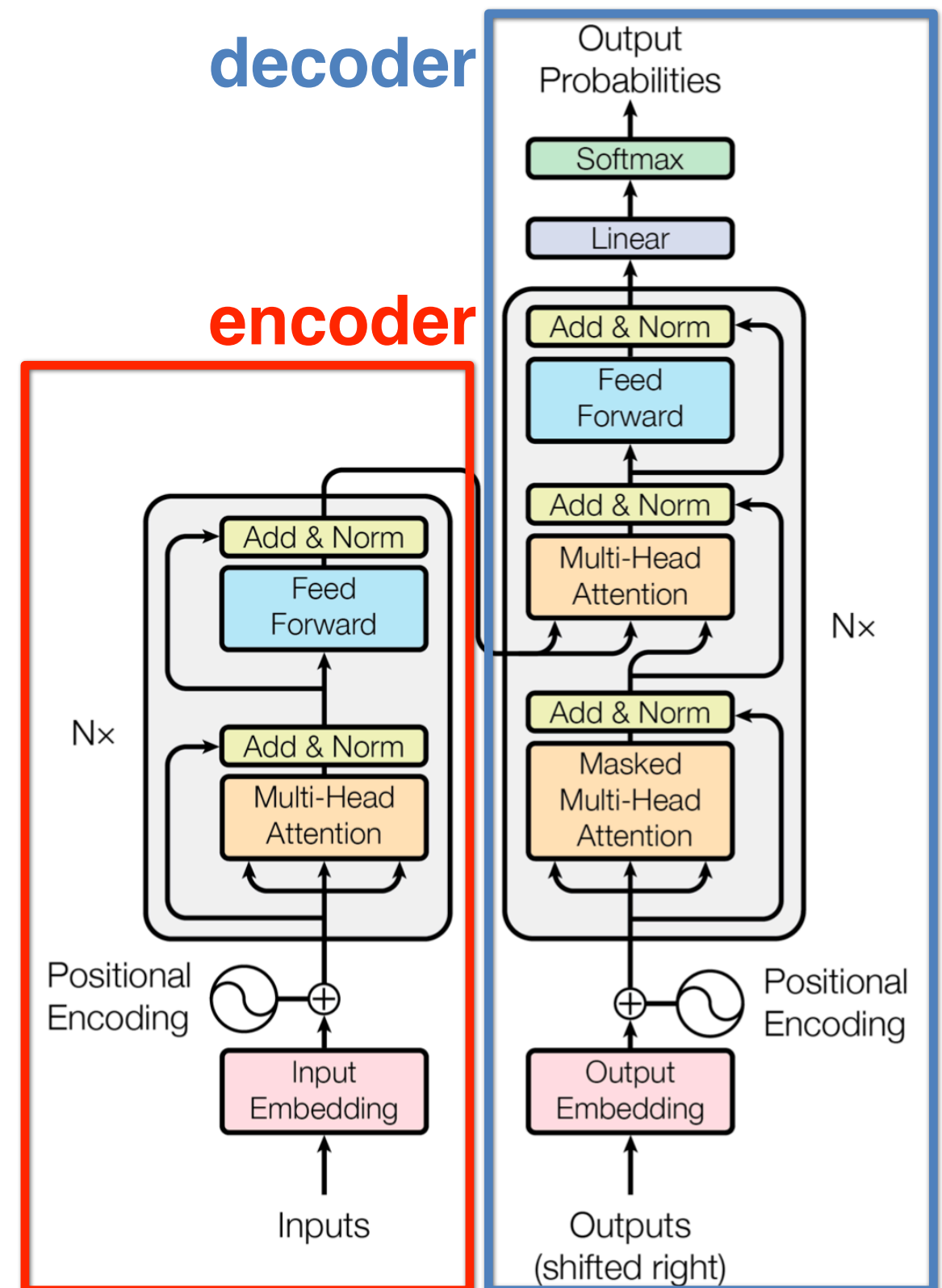


From GNN to Transformer

- Transformers have been historically developed for **translation** task: (<https://arxiv.org/abs/1706.03762>)
- Uses an **attention mechanism** instead of convolution or graph
 - ➔ Shorter connection between input, better gradient flow
- Huge success, nowadays used as a cornerstone for **Conversational Generative AI** (chatGPT...)
- I will first show the original architecture then discussed how it as evolved

The Transformer

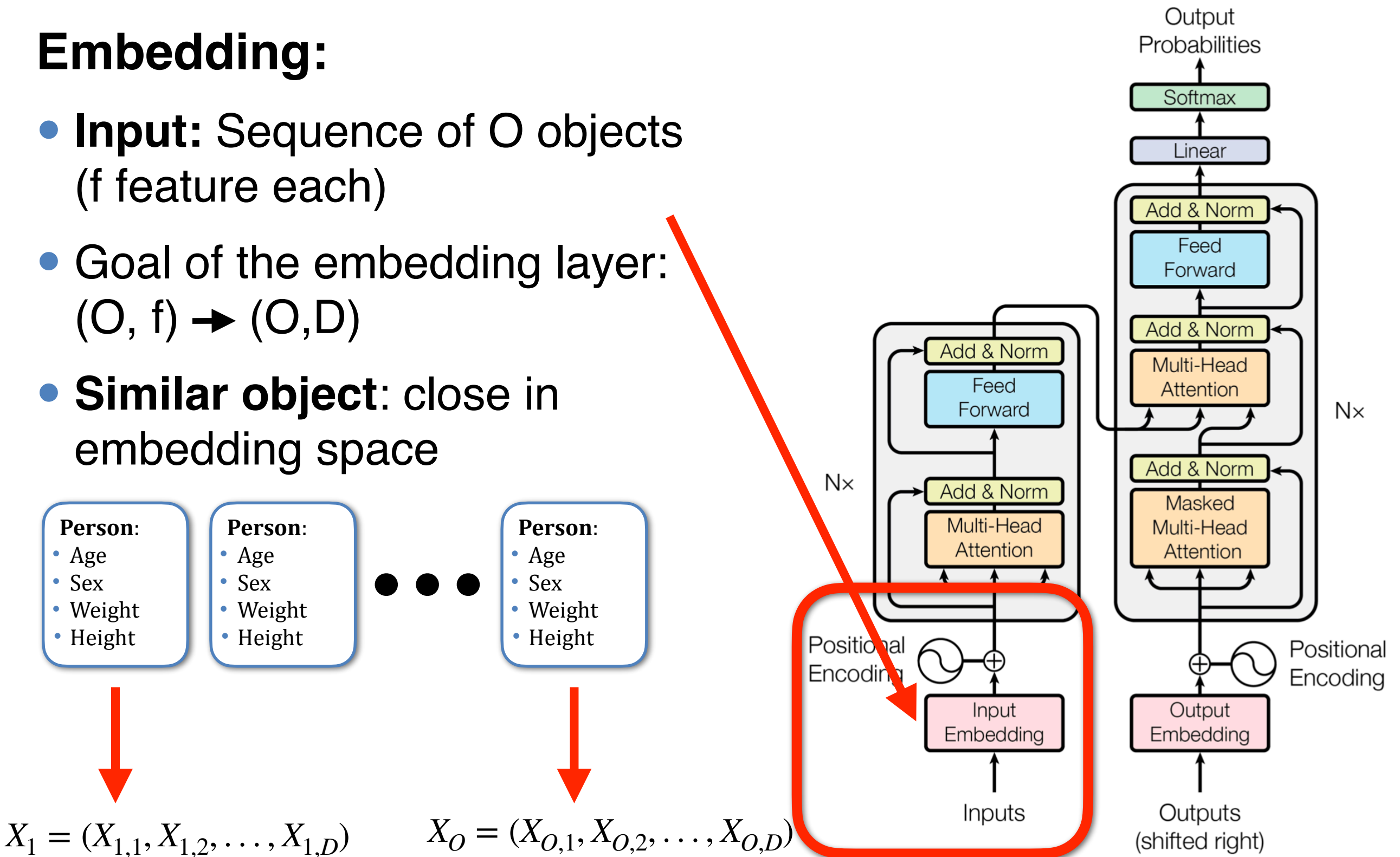
- Original architecture works with an **encoder/decoder** structure
- The encoder project the input sequence into an **latent space**
- The decoder then **generate** a sequence based on that input
- Not used this way anymore but useful to understand



The Transformer: Embedding

Embedding:

- **Input:** Sequence of O objects (f feature each)
- Goal of the embedding layer: $(O, f) \rightarrow (O, D)$
- **Similar object:** close in embedding space



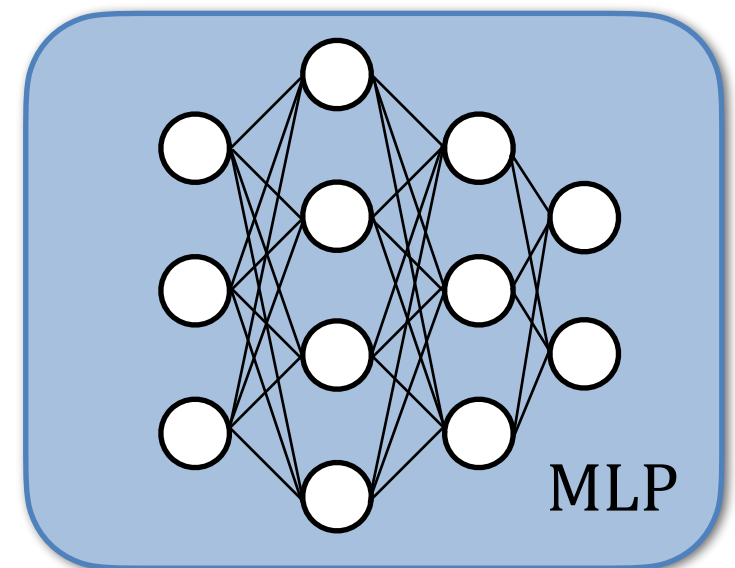
The Transformer: Embedding

Different approach possible:

- Discrete, known entry: Dictionary
- Continuous variable: MLP
- Other approaches based on domain knowledge (see tomorrow's seminar)

Dictionary:

- learn one vector per possible entry
- $W \in \mathbb{R}^{\text{dict_size} \times D}$
- Pre-trained for language



$$\text{Emb}(\text{pos}) = (\cos(2\pi \times \text{pos}), \sin(2\pi \times \text{pos}), \\ \cos(4\pi \times \text{pos}), \sin(4\pi \times \text{pos}), \\ \dots \\ \cos(2^k \pi \times \text{pos}), \sin(2^k \pi \times \text{pos}))$$

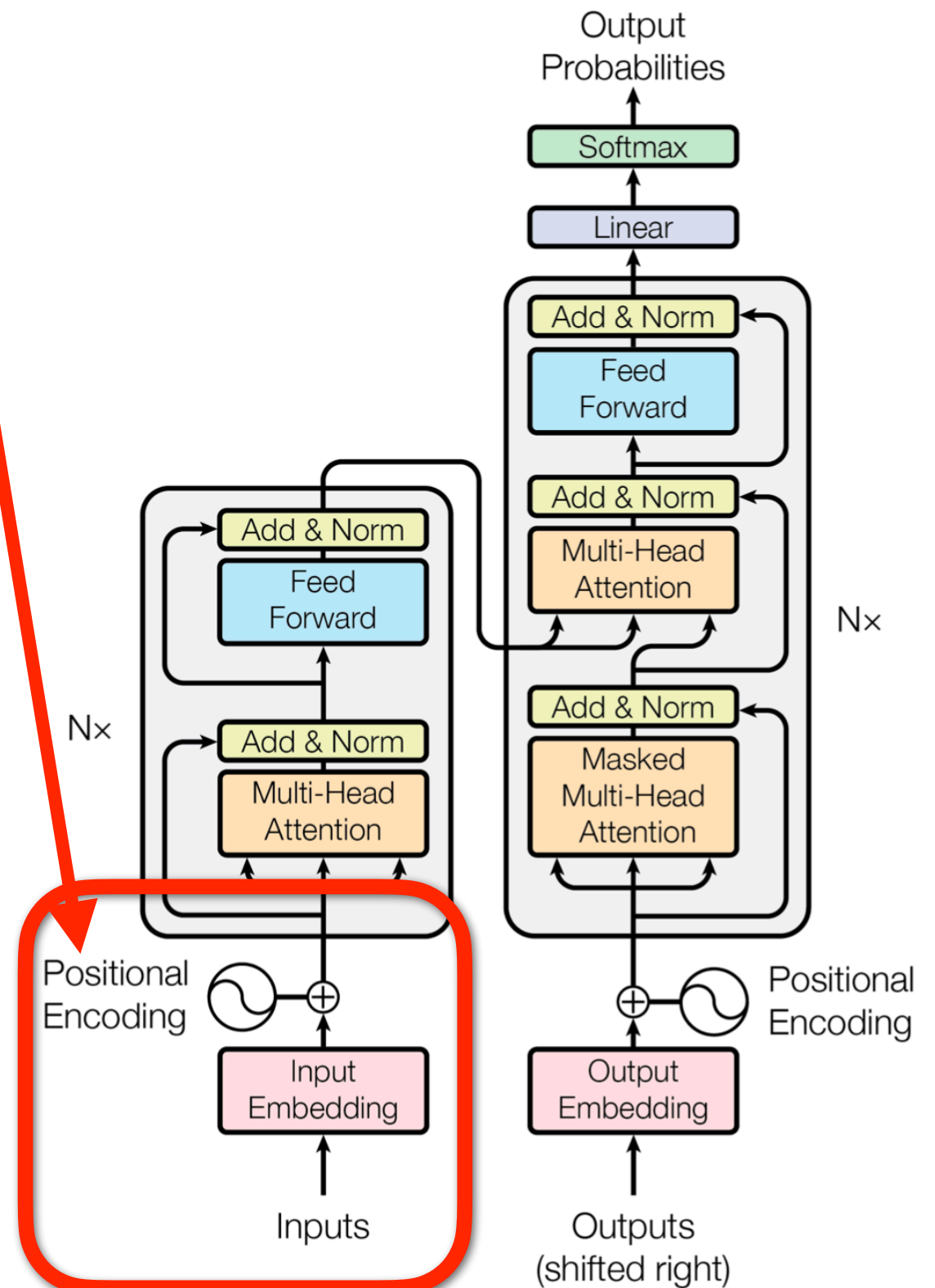
Fourier Embedding

The Transformer: Embedding

Optional step: **Positional Encoding**

- Transformer: **order invariant**
- Positional Encoding add positional information
- Modify the (O,D) matrix by adding a **sinusoidal** function to it
- Useful for **language**, less for physics

$$PE_{(pos, 2i)} = \sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos, 2i+1)} = \cos(pos/10000^{2i/d_{\text{model}}})$$



The Transformer: Attention

- Goal: Learn the **correlation** that exist between all the input
- Split the input embedding into 3 vector: **Queries, Keys, Values**

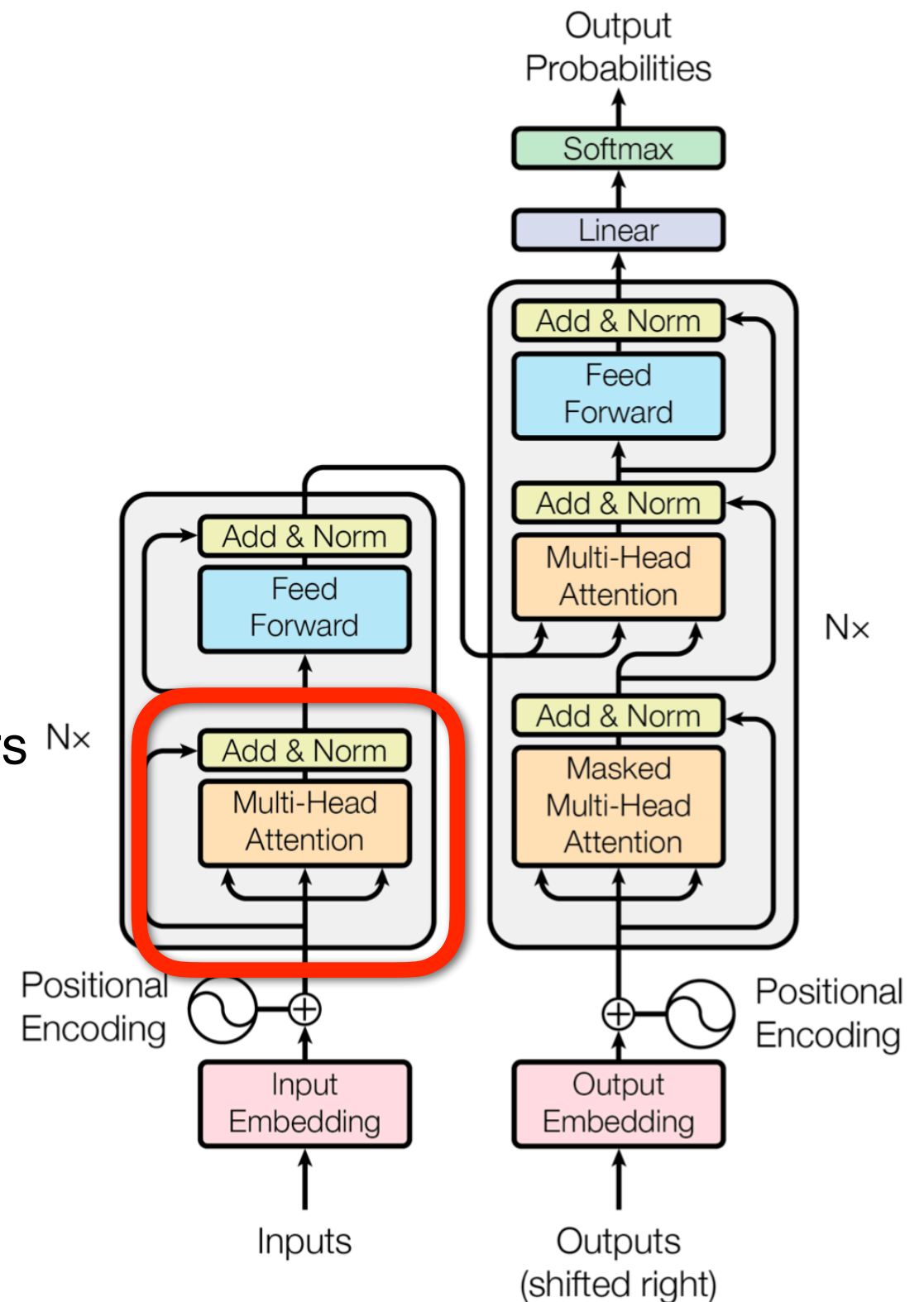
Embedding (single object): x_i

Learned parameters $N \times$

$$\text{Queries : } q_i = x_i \times W^Q$$

$$\text{Keys : } k_i = x_i \times W^K$$

$$\text{Values : } v_i = x_i \times W^V$$



The Transformer: Attention

Exemple for 2 input objects:

- $score1 = q_2 \times k_1$
- $score1 = score1 / \sqrt{d}$
- $score1 = SoftMax(score1) = 0.1$
- $score1 = score1.v_1$

- $score2 = q_2 \times k_2$
- $score2 = score2 / \sqrt{d}$
- $score2 = SoftMax(score2) = 0.9$
- $score2 = score2.v_2$

$$z_2 = \sum_{I=1}^N scoreI$$

Strong (self) correlation

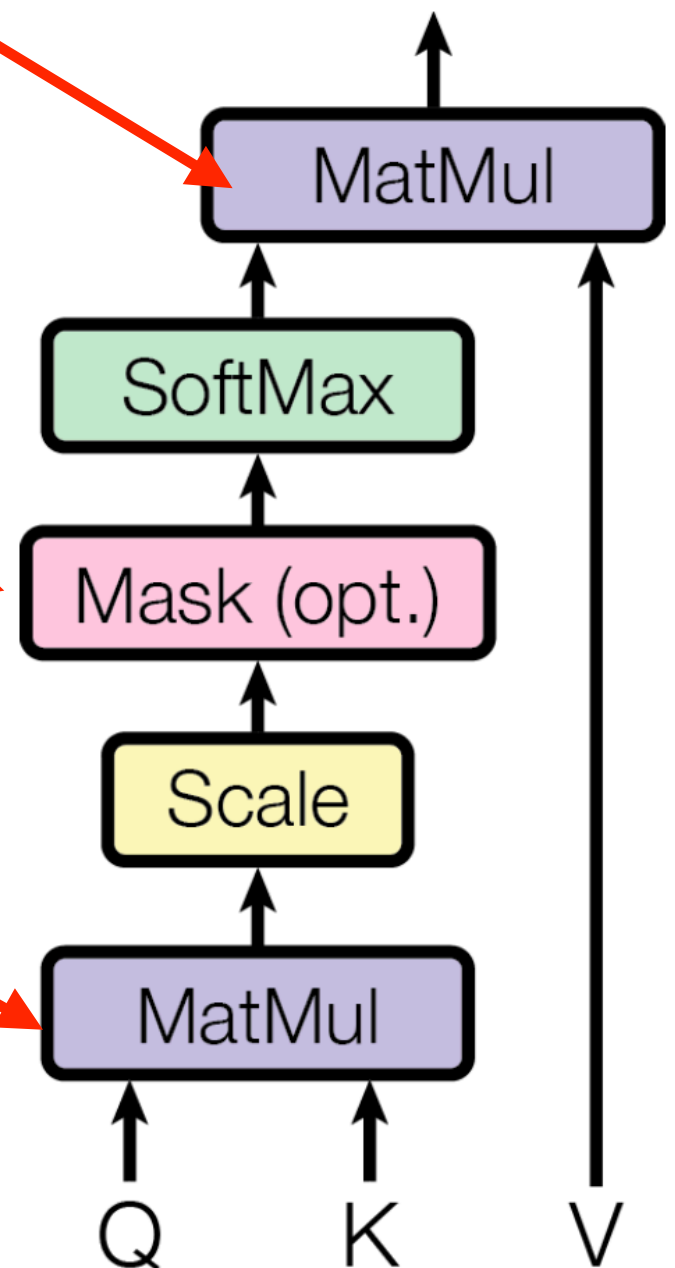
Weak correlation

Correlation $\times$ queried value

Output of size D

Allow of to ignore some pairs

Compare Query and Key



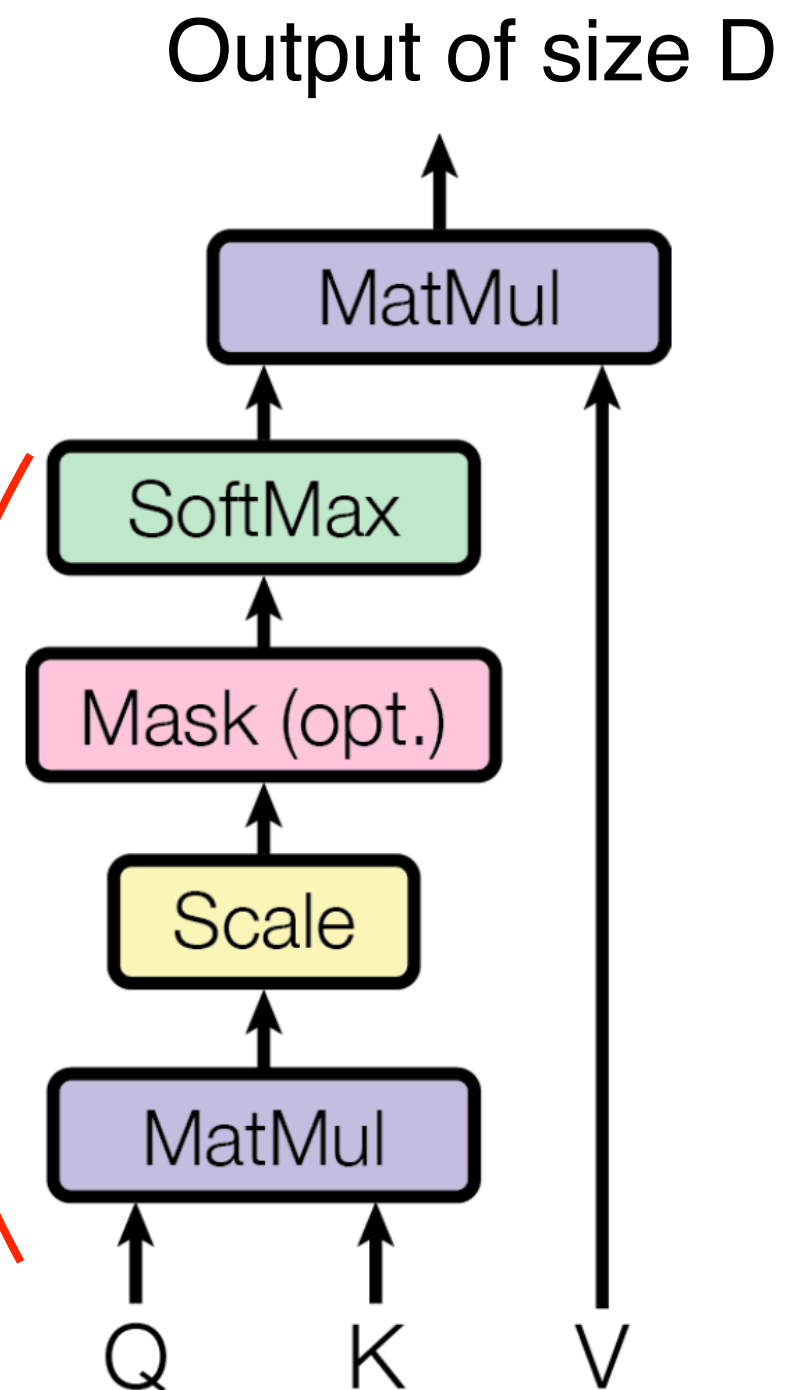
The Transformer: Attention

$$A = \begin{bmatrix} att_{11} & att_{12} & att_{13} & \dots & att_{1n} \\ att_{21} & att_{22} & att_{23} & \dots & att_{2n} \\ \dots & \dots & \dots & \dots & \dots \\ att_{n1} & att_{n2} & att_{n3} & \dots & att_{nn} \end{bmatrix}$$

Build an **attention matrix**
~ input correlation

$$Out = \left[\sum (att_{i1} \times V_i) \quad \sum (att_{i2} \times V_i) \quad \dots \quad \sum (att_{in} \times V_i) \right]$$

Output based on **attention** and **value keys**



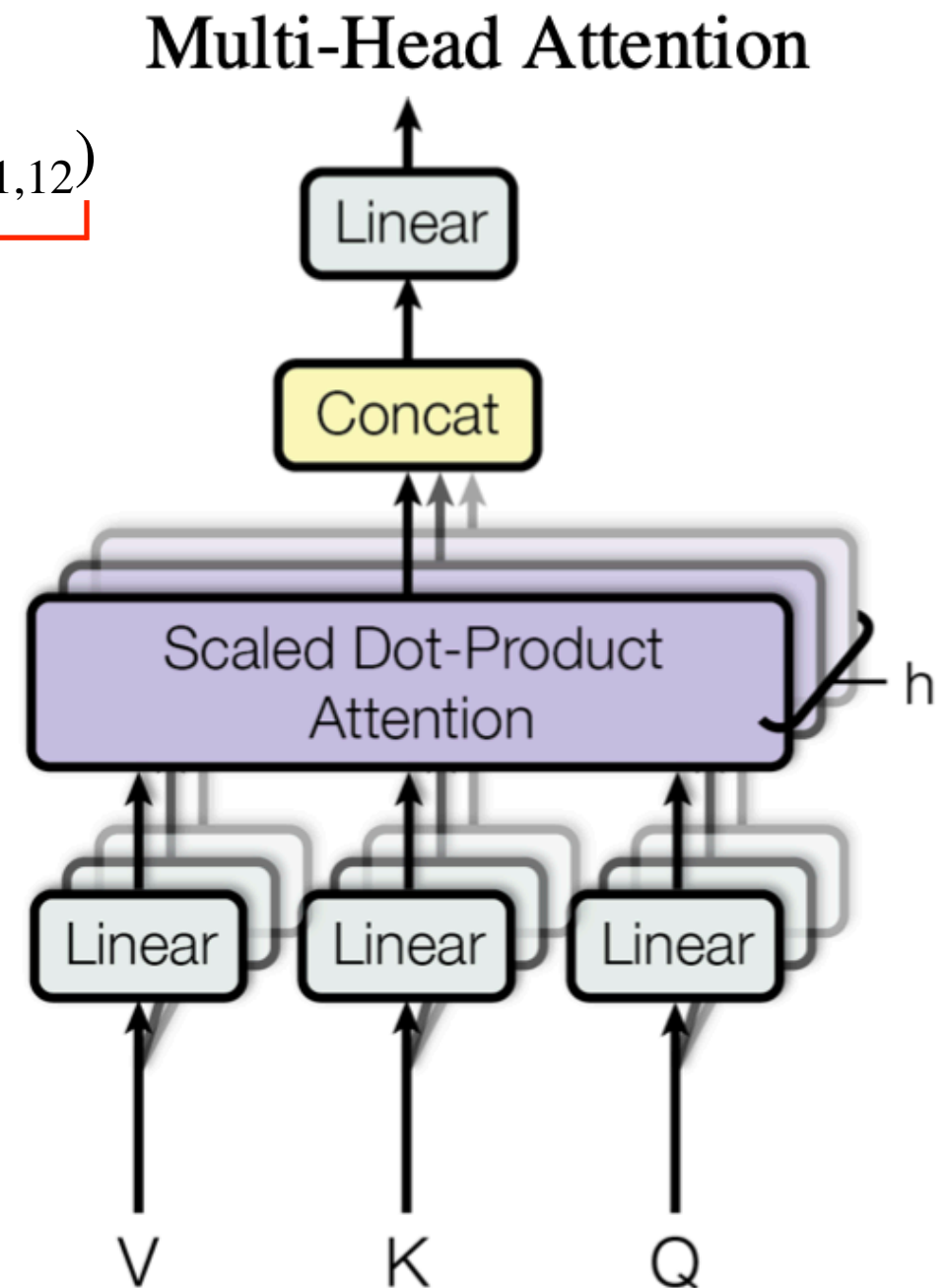
The Transformer: Attention

- Instead of building 1 attention we can build multiple ones in parallel

$(X_{1,1}, X_{1,2}, X_{1,3}, X_{1,4}, X_{1,5}, X_{1,6}, X_{1,7}, X_{1,8}, X_{1,9}, X_{1,10}, X_{1,11}, X_{1,12})$

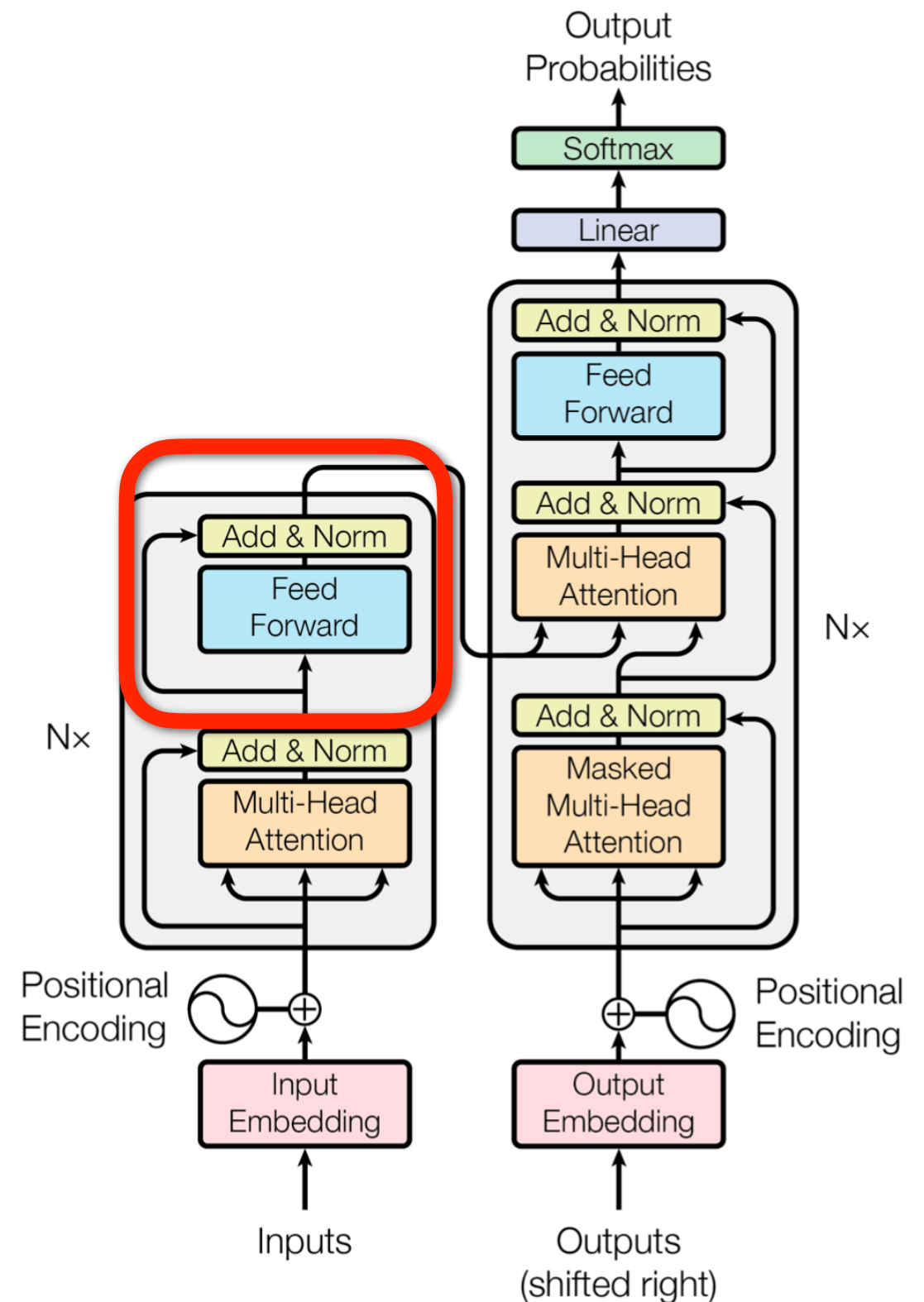
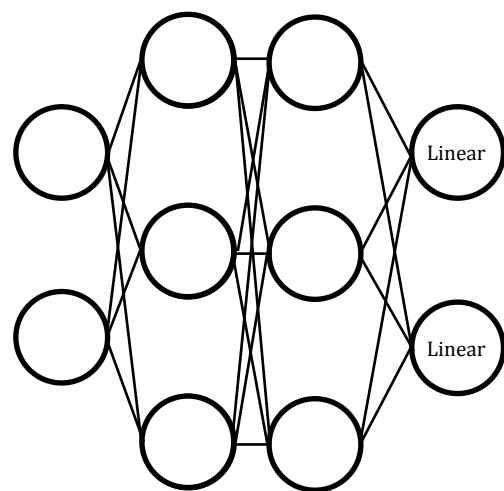
head 1 head 2 head 3

- Concatenate the result of all head afterward
- Useful in the context of positional encoding, as each attention head can look at a different "scale"
- Equivalent to a single attention head in term of computation



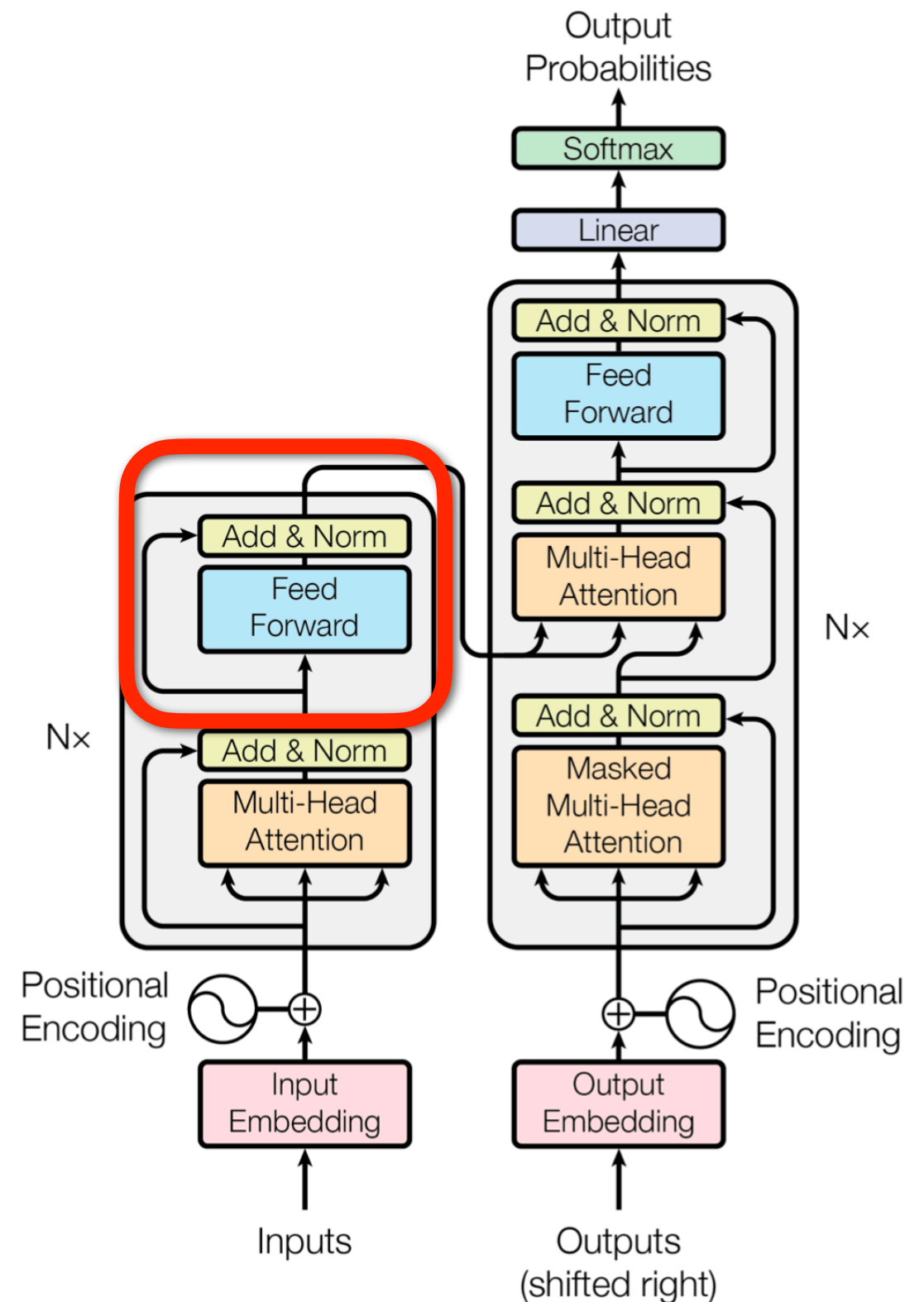
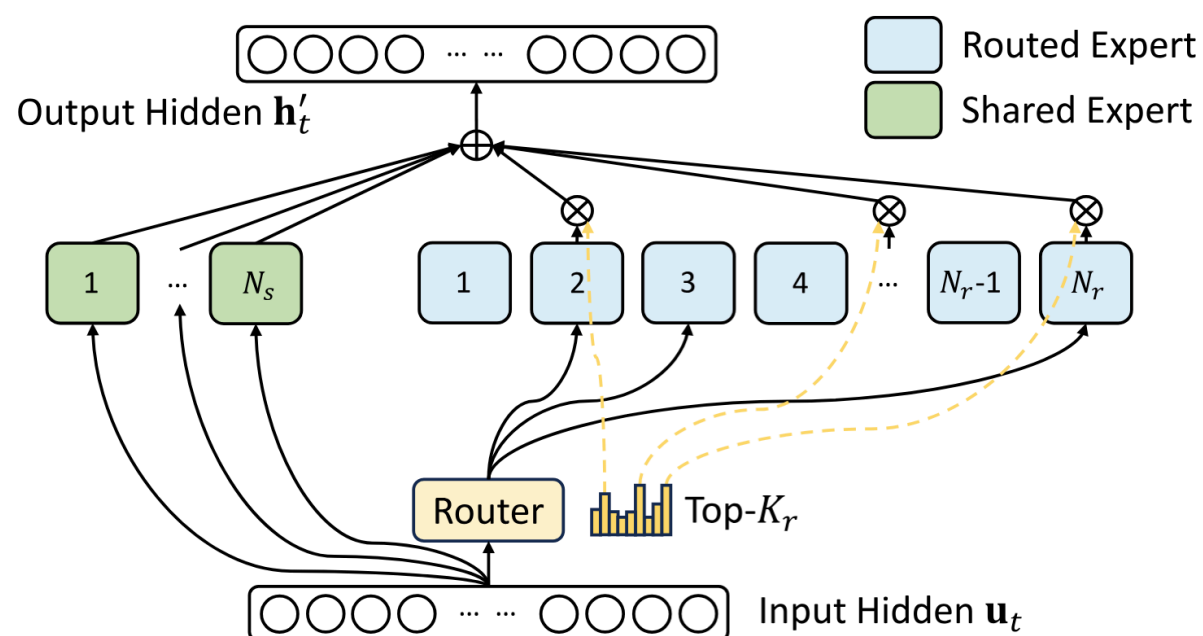
The Transformer: Feed Forward

- Attention: $(O,D) \rightarrow (O,D)$
- We are learning **correlation** but not computing anything new !
- Feed Forward network = 2 layer MLP
- Extract information for the new embedding of each object
- Applied to **each element** of the sequence separately



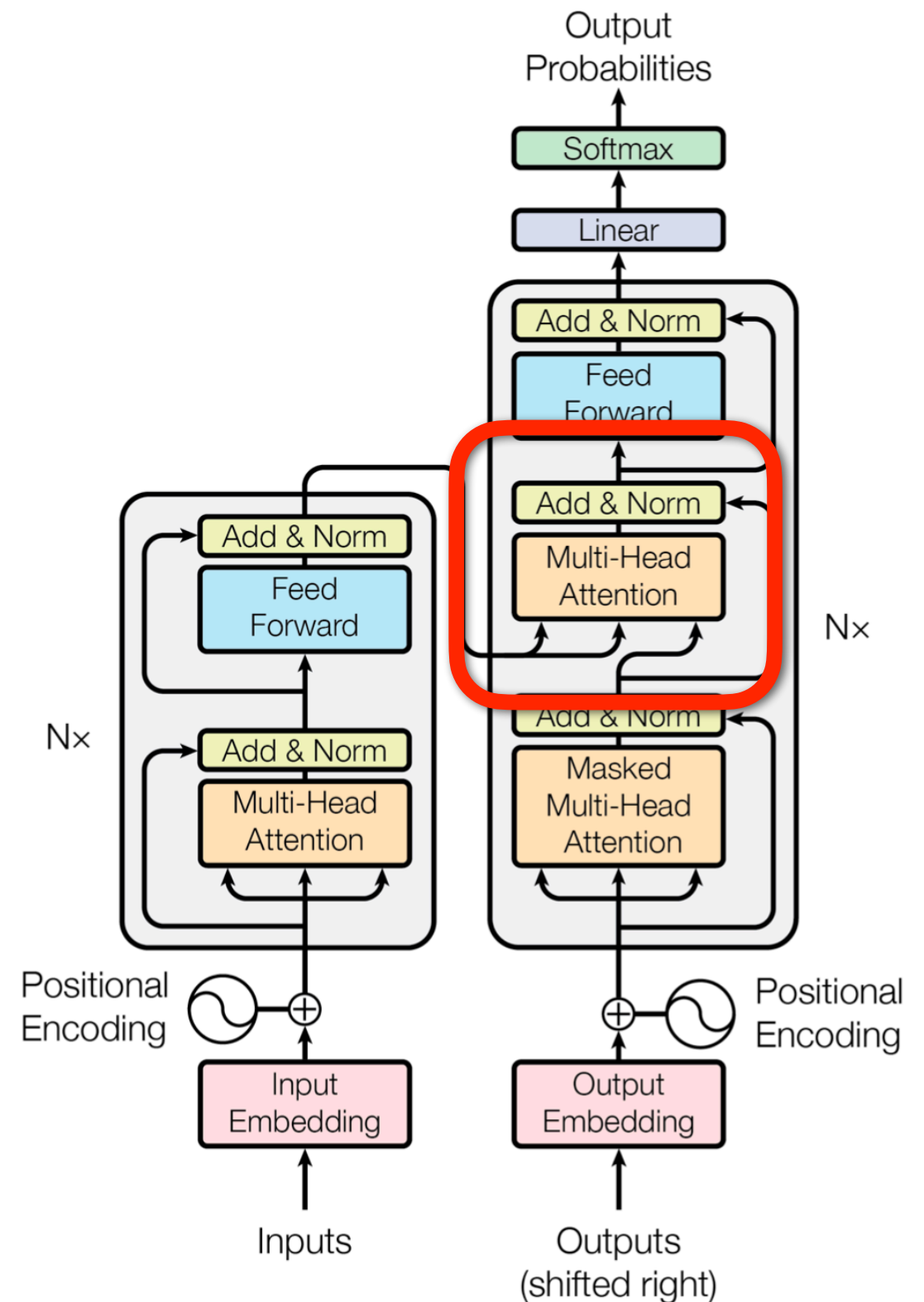
The Transformer: Mixture of expert

- Most of the weights of a transformer are in the FF
- Very large network! (**Billions** of weight)
- One solution: **MoE**
- Many different networks, choose one based on the input value



The Transformer: Decoder

- Act as the **generative** part of the network
- Start from an empty sequence, try to generate the next element
- Same architecture as encoder with an extra block:
 - ➔ Attention head that use key from the **new sequence** and query and value for the **encoder**
- Loop until the **output sequence** is completed

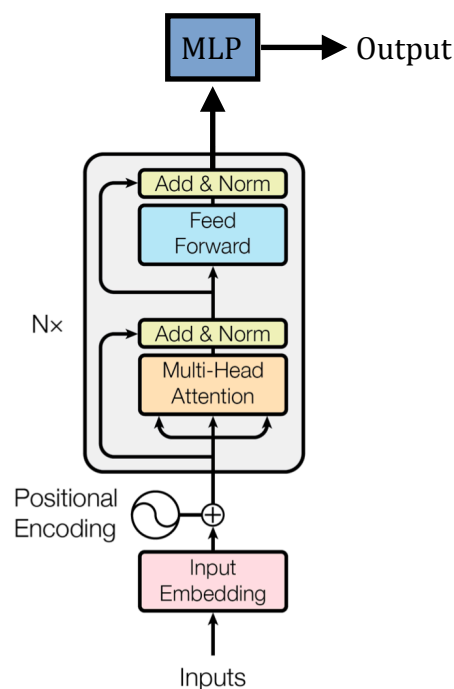


The Modern Transformer

Transformer task can be split into 2 categories :

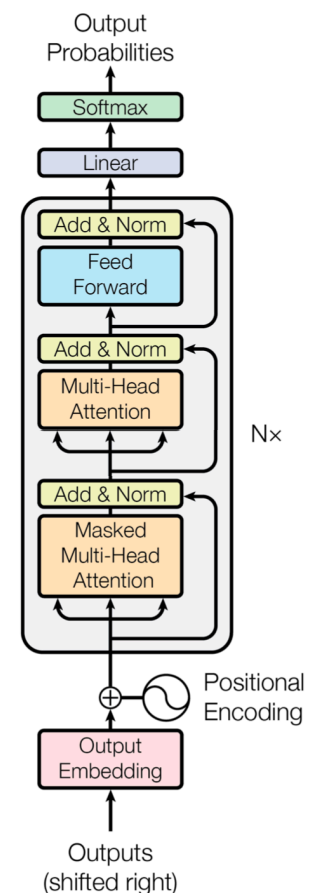
Regression/Classification:

- Use Encoder alone
- At the end of the transformer a MLP compute the target variable/score



Generation :

- Use the decoder alone
- The context is no longer provided by the encoder but directly through the decoder input



- Easy to rewrite or load from your favorite DL library

Building your own Transformer

- Most **components** needed to build a transformer are available in PyTorch
- Embedding dictionary: `torch.nn.Embedding`
- Encoder: `torch.nn.TransformerEncoder`
- Easy to write yourself, I recommend the exercise at least once
- Be careful if you do, **export** might be very inefficient

```
def manual_attention(q: Tensor, k: Tensor, mask: Optional[Tensor] = None) -> Tensor:
    """Compute the attention matrix manually

    Args:
        q: Query tensor of shape (batch_size, num_heads, seq_len, d_k)
        k: Key tensor of shape (batch_size, num_heads, seq_len, d_k)
        mask: Optional mask tensor of shape (batch_size, num_heads, seq_len, seq_len)

    Returns:
        Tuple of (output tensor, attention weights matrix on the last attention layer)
    """
    d_k = q.size(-1)
    scores = torch.matmul(q, k.transpose(-2, -1))
    scale = q.new_full((), d_k).sqrt()
    scores = scores * scale.reciprocal()

    if mask is not None:
        scores = scores.masked_fill(mask, float("-inf"))
    score = F.softmax(scores, dim=-1)

    return score
```

Attention optimisation

Attention mechanism:

- nb_head matrices of size (O, O)
- Large amount of computation
 - ➔ ~50% of the inference time of the transformer
- Huge memory footprint
- Scale as $O(n^2)$
- Many different **implementations** try to improve this

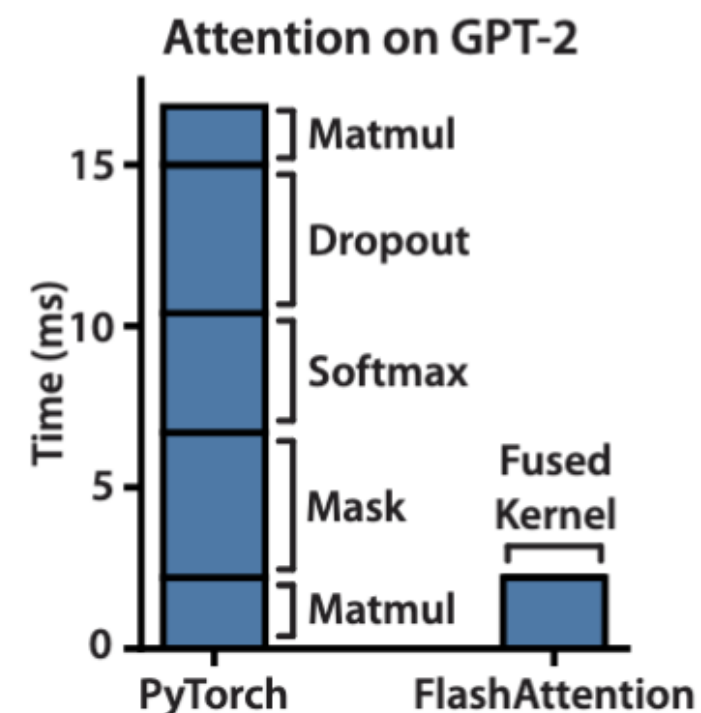
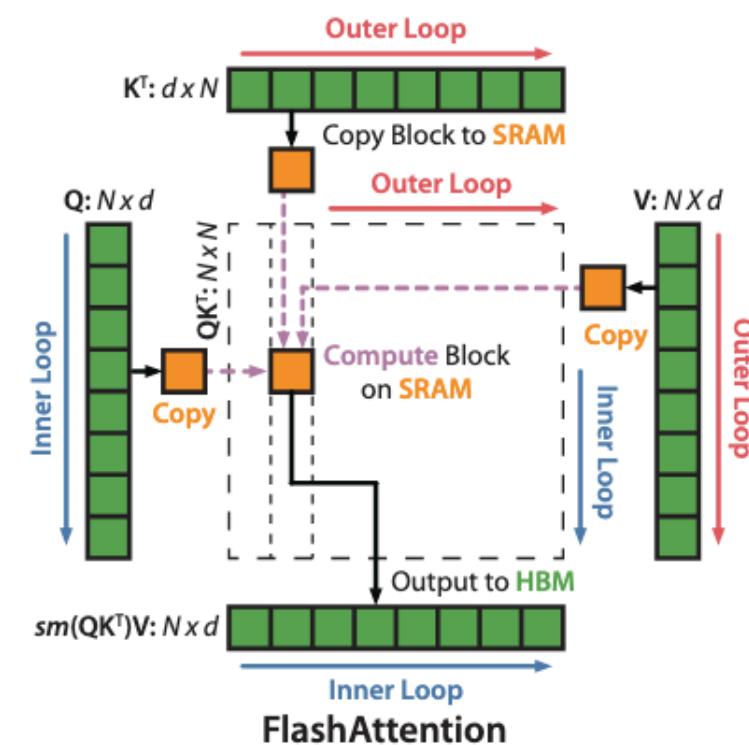
Flash Attention

Attention mechanism: h matrices of size (O, O)

- Large amount of **computation**
- Huge **memory** footprint

Solution: **FlashAttention**

- Nowadays available by default in PyTorch
- Perform the matrix multiplication by block
- Optimise GPU I/O operation
- Mathematically equivalent but can be used on longer sequences



<https://arxiv.org/pdf/2205.14135>

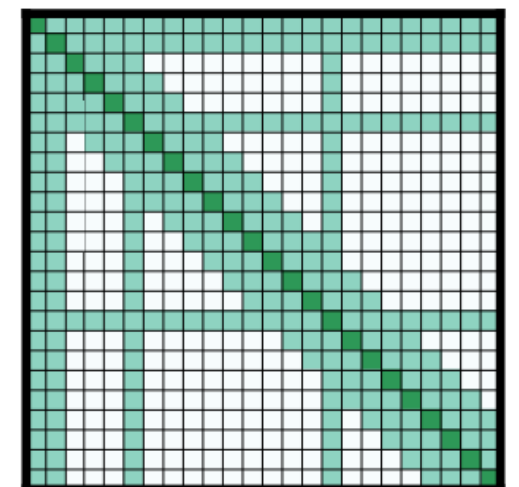
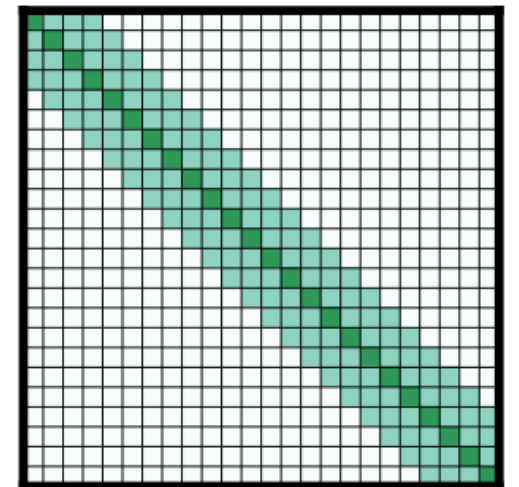
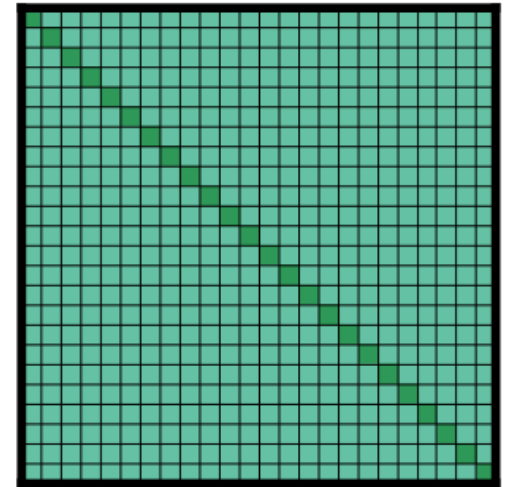
Sparser Attention

Attention mechanism: h matrices of size (O, O)

- Full correlation might not be needed
- Using **sparse matrices** can improve computation

Many different sparse implementations:

- [Longformer](#)
- [Big Bird](#)
- ...
- Requires you to be able to know which pairs of entries are uncorrelated



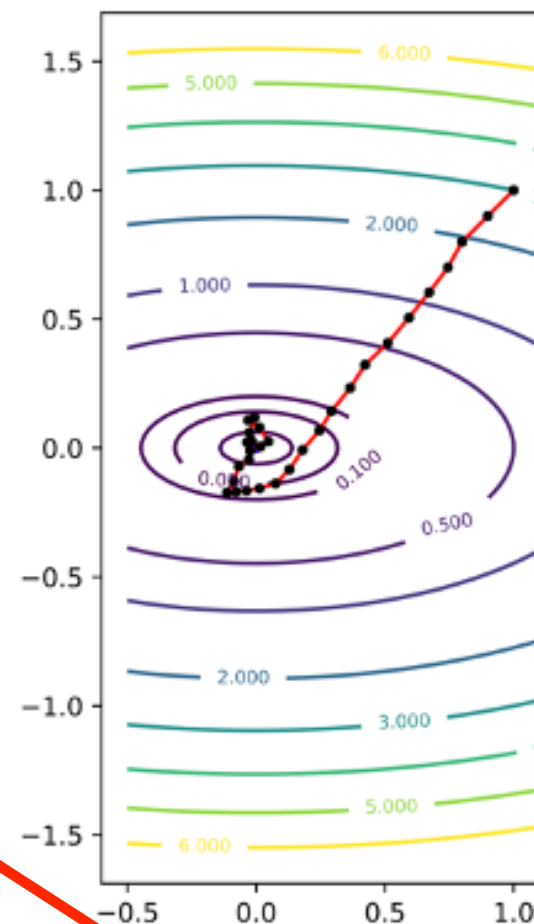
A few pointers on using Transformers

$$W_{t+1} = W_t - \alpha \nabla Loss$$

- Transformers: large number of **free parameters** ($\sim M$)
- Proper choice of the **optimiser** and **learning Rate** important to achieve good performances
- **AdamW** has been shown to perform well; other options exist (such as Lion, might be faster ?)
- Uses weight decay ($\lambda = 0.01$) to force weights to stay around to 0

Depends on the optimiser

Adam



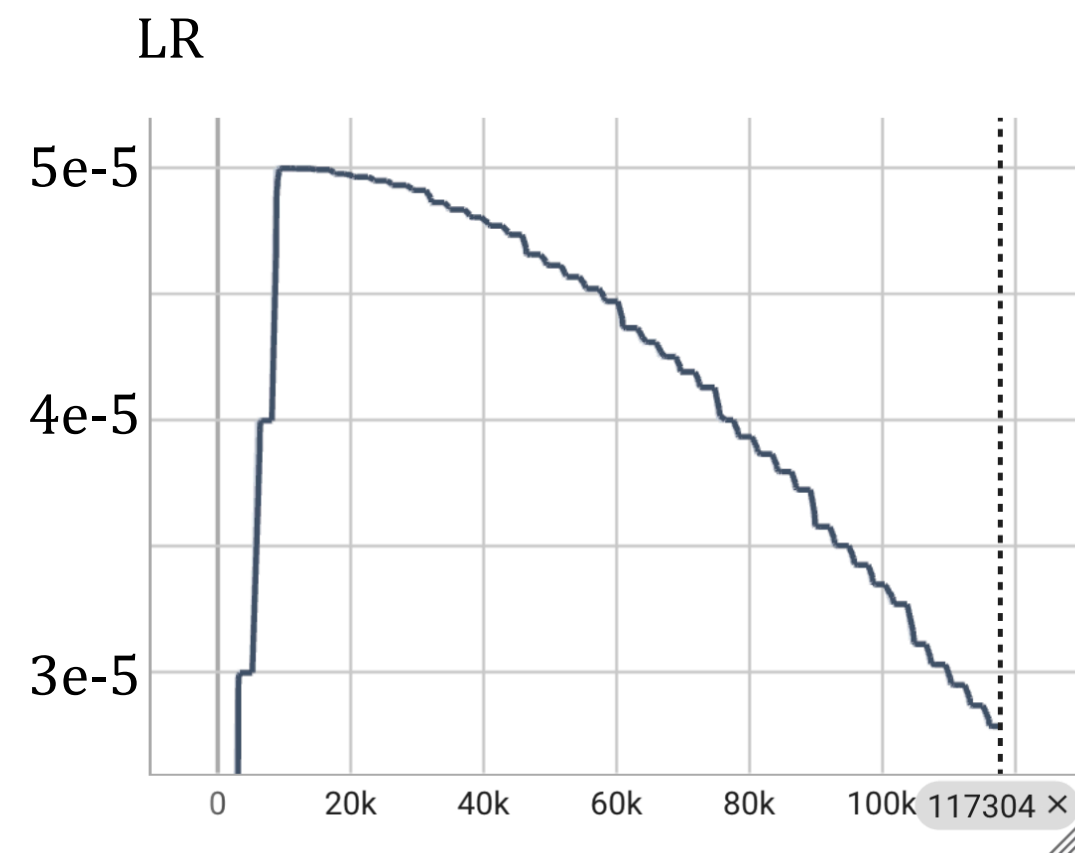
$$W_{t+1} = W_t - \alpha (\nabla Loss - \lambda W_t)$$

A few pointers on using Transformers

$$W_{t+1} = W_t - \alpha \nabla Loss$$

Learning rate

- Learning rate control how much we want to **vary** the network weights at a given step
- Need to be **kept low** when training transformer ($\sim e-5$) -> smoother gradient
- Learning rate with warmup are popular nowadays
 - ➔ Avoid overtraining on the first sample
 - ➔ Ensure the network keep specialising

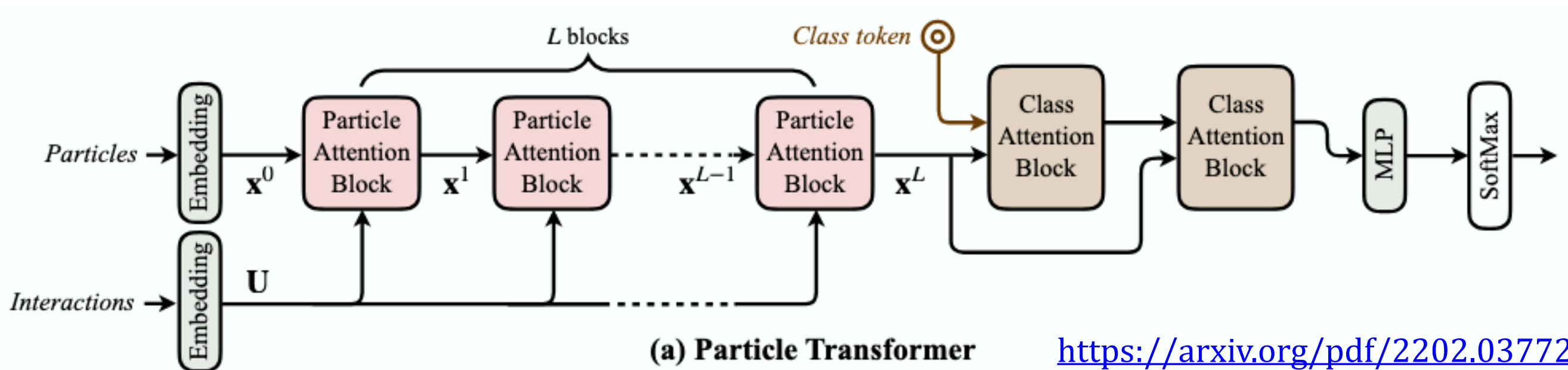


Examples: CMS Particle Transformer

Particle Transformer:

- CMS approach to jet tagging
- Custom network architecture
-> take advantage of physics knowledge
- Two input type:
 - Particles: C features each
 - Interaction: C' features for each particle pairs

$$\Delta = \sqrt{(y_a - y_b)^2 + (\phi_a - \phi_b)^2},$$
$$k_T = \min(p_{T,a}, p_{T,b}) \Delta,$$
$$z = \min(p_{T,a}, p_{T,b}) / (p_{T,a} + p_{T,b}),$$
$$m^2 = (E_a + E_b)^2 - \|\mathbf{p}_a + \mathbf{p}_b\|^2,$$

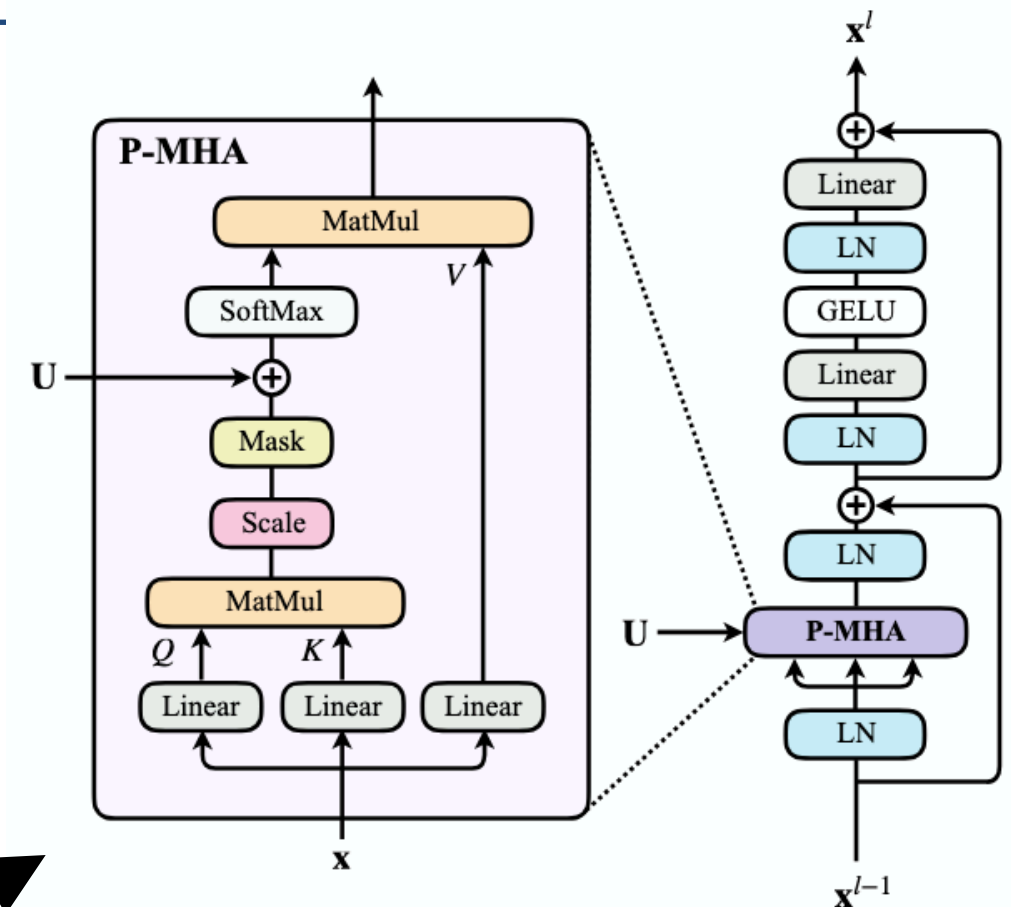


<https://arxiv.org/pdf/2202.03772>

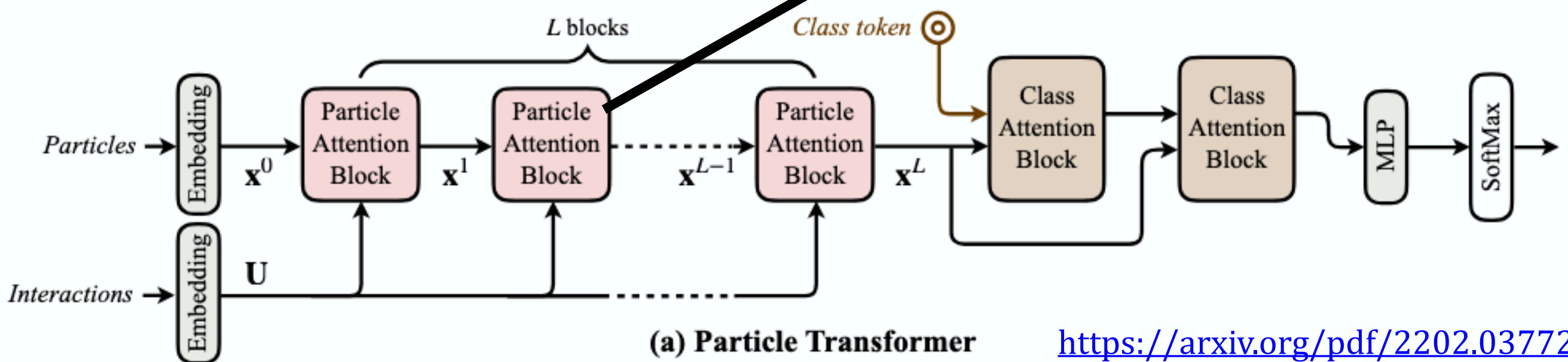
Examples: CMS Particle Transformer

Particle Attention Block:

- Particle embedding ($N, 128$)
- Interaction embedding ($N, N, 8$)
- Bias the attention matrix with the interaction U (1dim per head)
- Take advantage of known cross-particles features



(b) Particle Attention Block



(a) Particle Transformer

<https://arxiv.org/pdf/2202.03772>

Examples: CMS Particle Transformer

Class attention:

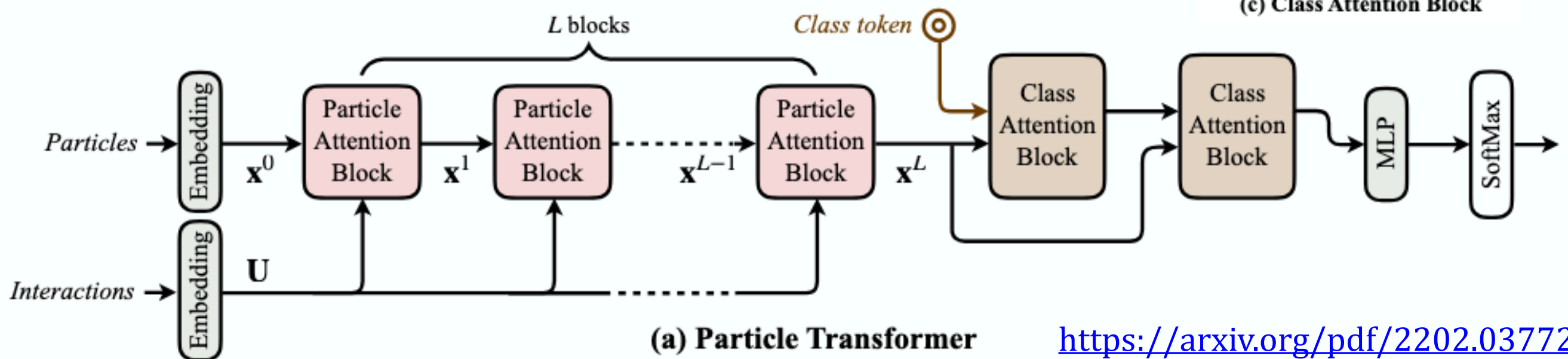
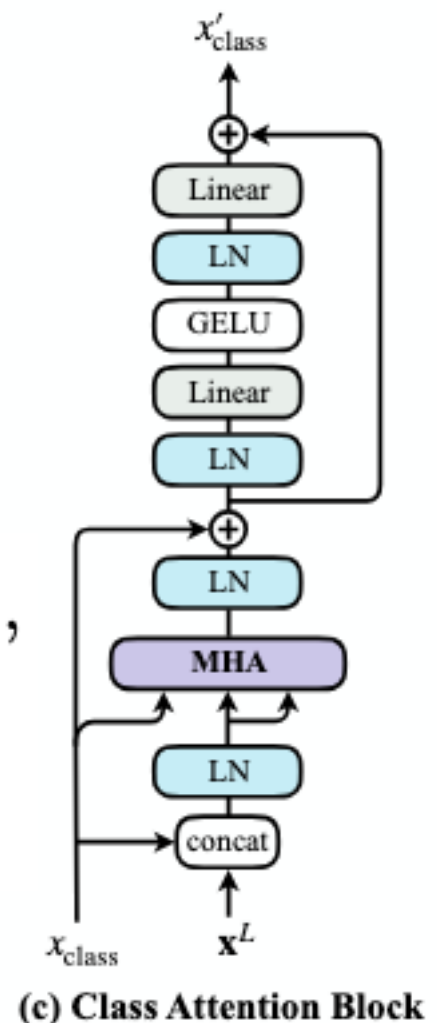
- Attention between a class token and the different particles
- We then get a score for each class

- $H \rightarrow b\bar{b}$
- $H \rightarrow c\bar{c}$
- $H \rightarrow gg$
- $H \rightarrow 4q$
- $H \rightarrow l\nu qq'$
- $t \rightarrow bqq'$
- $t \rightarrow bl\nu$
- $W \rightarrow qq'$
- $Z \rightarrow qq'$

$$Q = W_q x_{\text{class}} + b_q,$$

$$K = W_k \mathbf{z} + b_k,$$

$$V = W_v \mathbf{z} + b_v,$$



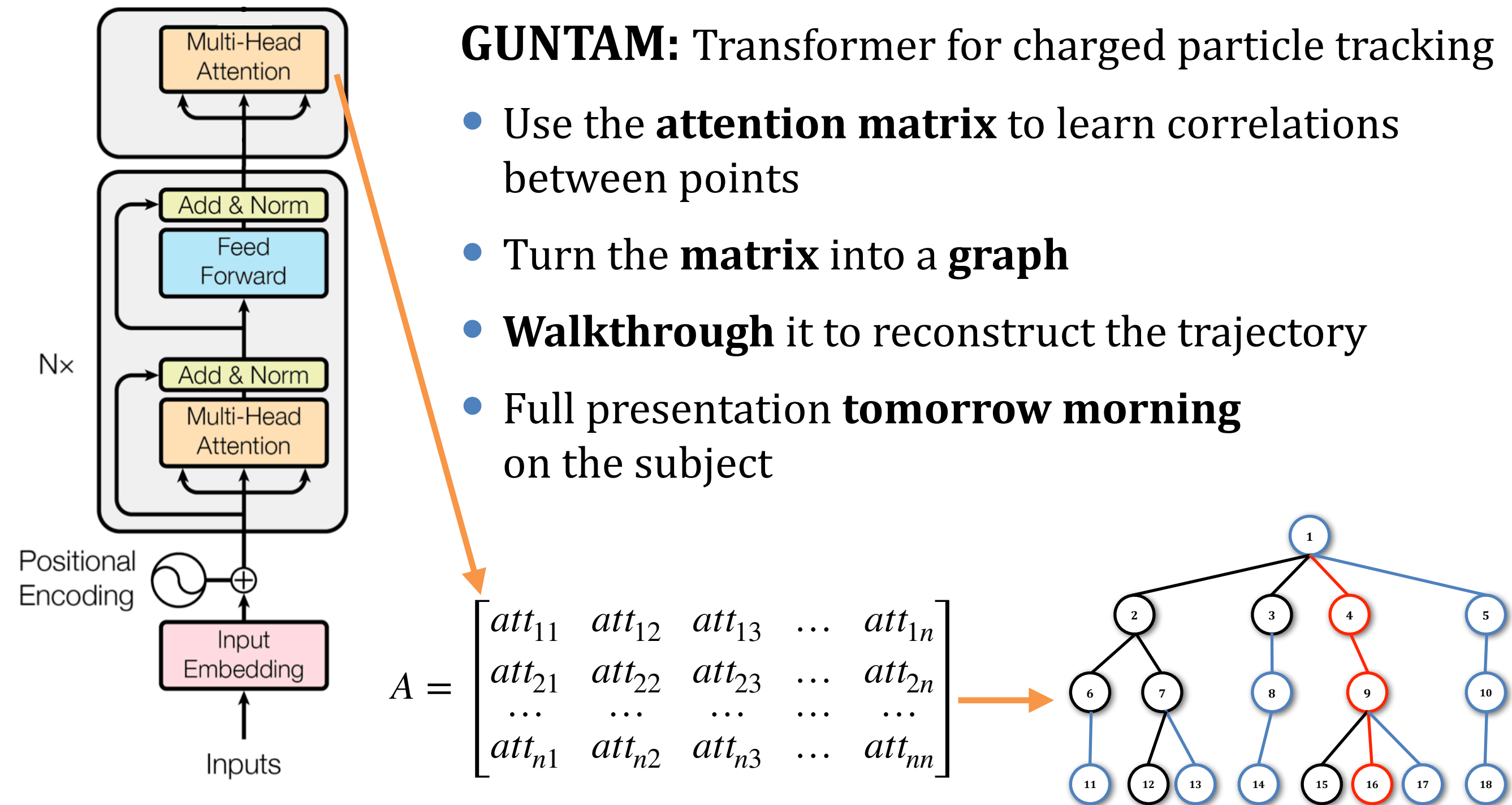
<https://arxiv.org/pdf/2202.03772>

Examples: GUNTAM

Generic
Unilateral
Neural
Track
Approximator
from **M**easurement

GUNTAM: Transformer for charged particle tracking

- Use the **attention matrix** to learn correlations between points
- Turn the **matrix** into a **graph**
- **Walkthrough** it to reconstruct the trajectory
- Full presentation **tomorrow morning** on the subject



Conclusion

- Transformers are becoming prevalent in HEP **reconstruction/tagging** tasks
- The right tool when operating on an **arbitrary number of input** elements
- Requires a large amount of computing resources and training data
- Tomorrow, a follow-up on GUNTAM: **a Transformer for charged particle tracking**

Thank you

Generic
Unilateral
Neural
Track
Approximator
from **M**easurement

